UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ROGÉRIO PALUDO

METODOLOGIA PARA VERIFICAÇÃO FUNCIONAL ANTECIPADA DE SOFTWARE EMBARCADO COMBINANDO PLATAFORMAS VIRTUAIS E VERIFICAÇÃO FORMAL

Florianópolis
2016

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Paludo, Rogério
Metodologia Para Verificação Funcional Antecipada De Software Embarcado Combinando Plataformas Virtuais e Verificação Formal / Rogério Paludo ; orientador, Djones Lettnin - Florianópolis, SC, 2016.
131 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia Elétrica.

Inclui referências

1. Engenharia Elétrica. 2. Verificação Software Embarcado. 3. Verificação Formal. 4. Simulação. 5. Plataformas Virtuais. I. Lettnin, Djones. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

Rogério Paludo

Metodologia Para Verificação Funcional Antecipada De Software Embarcado Combinando Plataformas Virtuais e Verificação Formal

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica para Obtenção do Grau de Mestre em Engenharia Elétrica.

Universidade Federal de Santa Catarina

Orientador: Prof. Dr. Djones Vinicius Lettnin

Florianópolis
28 de março de 2016

ROGÉRIO PALUDO

METODOLOGIA PARA VERIFICAÇÃO FUNCIONAL ANTECIPADA DE SOFTWARE EMBARCADO COMBINANDO PLATAFORMAS VIRTUAIS E VERIFICAÇÃO FORMAL

Esta Dissertação foi julgada e aprovada para a obtenção do Título de "Mestre em Engenharia Elétrica", e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica.

Florianópolis, 9 de Março de 2016.

Prof. Dr. Carlos Galup-Montoro
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Djones Vinicius Lettnin
Orientador

Prof. Dr. Lucas Carvalho Cordeiro
Universidade Federal do Amazonas

Profª. Drª. Karina Rocha Gomes da Silva
Universidade Federal de Goiás

Prof. Dr. Raimes Moraes
Universidade Federal de Santa Catarina

Às três pessoas mais importantes na minha vida: meus pais, Leonyr e João e a minha esposa Patrícia.

AGRADECIMENTOS

Diversas pessoas tiveram contribuição significativa neste trabalho, para evitar esquecimentos ou desmerecimentos, não irei explicitá-las aqui. Porém, irei honrá-las com uma frase célebre de Isaac Newton:

"If I have seen further it is by standing on the shoulders of giants."

“Elvis has Spirit. The answer is 42...END\r\n;”
Trecho da mensagem transmitida pela
Curiosity quando não há dados de
telemetria para serem enviados à Terra.

RESUMO

O crescente volume e complexidade de *software* sendo utilizado em aplicações embarcadas introduz novos desafios para verificação. Além disso, cada vez mais sistemas controlados por *software* são inseridos diariamente nas nossas vidas, criando novas formas de interação e trazendo preocupações gradativas quanto integridade. Esse cenário pode ser observado pelo recente número de padrões destinados a fornecer mecanismos de segurança funcional, como exemplos os padrões ISO 26262 na área automotiva, IEC 61513 na área de geração de energia e IEC 62304 na assistência médica. Percebe-se que muitos sistemas que antes não eram tratados como críticos, devem ser desenvolvidos e verificados de tal forma atualmente. Associado a esse ponto de vista técnico, o mercado atual demanda alta produtividade e reduzido *time-to-market*. Assim, são necessárias alternativas que forneçam suporte ao desenvolvimento de *software* embarcado, considerando verificação ainda em fases iniciais do projeto. É importante perceber que isso não é somente uma exigência do mercado, pois a quantidade de erros de implementação introduzidos é muito maior durante a programação dos dispositivos do que em fases de especificação e elaboração. Levando em conta essas características, este trabalho expõe uma metodologia de desenvolvimento de *software* embarcado voltado para verificação nas fases iniciais de projeto, considerando ferramentas e abordagens atuais. Por parte de desenvolvimento são consideradas plataformas virtuais de simulação do sistema, as quais fornecem suporte para desenvolvimento mesmo antes do *hardware* final estar disponível. Essas mesmas plataformas permitem simulação de *software* dependente de *hardware* através de camadas de isolamento e modelagem de periféricos. Como a criação de plataformas virtuais é uma tarefa árdua, a linguagem de descrição de arquiteturas ArchC é utilizada para fornecer suporte a implementação de simuladores de conjunto de instruções. Do ponto de vista de verificação são utilizados métodos estáticos (i.e., *Model Checking*), para exploração de erros de implementação e verificação funcional com propriedades temporais. No entanto, apesar dos recentes avanços em *Model Checking*, limitações com relação a complexidade podem comprometer a verificação de sistemas complexos. Nesses casos, simulações e testes do sistema são conduzidos, através de plataformas virtuais, para obter maior cobertura e estresse do sistema, além é claro de fornecer informações valiosas quanto ao seu comportamento. Como resultados é demonstrado: o desenvolvimento e a verificação de um modelo baseado no microcontrolador MSP430; dois cenários de verificação híbrida de um sistema de controle de injeção de combustível; uma plataforma virtual de simulação de um sistema de controle mecânico, considerando modelos físicos integrados; e por fim, a especificação, implementação e teste de um computador de bordo de um CubeSat, um sistema

consideravelmente complexo, constituído de três unidades de processamento e com um sistema operacional de tempo real. Esses resultados servem como demonstração do potencial da metodologia e evidenciam a importância de verificação nas fases iniciais de projeto.

Palavras-Chave: *Model Checking*, Plataformas Virtuais, *software* embarcado, verificação antecipada.

ABSTRACT

The growing size and complexity of software being used in embedded applications introduce new verification challenges. Moreover, software-controlled systems are being inserted more and more into our daily routines, causing new forms of interaction and producing frequently integrity concerns. This outline is noticeable in the recent number of standards intended to provide functional safety mechanisms, examples are the ISO 26262 standard in the automotive industry, IEC 61513 for power generation and IEC 62304 in health care. One can see that many systems that were not treated as critical before must be treated similarly in the current situation. Associated with this technical point of view, the current market demands high productivity and reduced time-to-market. Thus, alternatives are required to provide support for the development of embedded software, considering verification even in early design stages of the project. It is important to realize that this is not only a market demand, the amount of errors of implementation introduced during programming is much higher than in specification and conceptual design. Given these aspects, this work presents an embedded software development methodology, focused on early verification considering current tools and approaches. On the development point of view, full system simulation is achieved through virtual platforms, which provide support for the development even before the final hardware is accessible. These same platforms enable simulation of hardware dependent software on isolation layers and model of the system peripherals. As virtual platform development can be a daunting task, the ArchC architecture description language is used to support the implementation of the instruction set simulators. On the verification viewpoint, static methods (i.e., Model Checking) are used to explore implementation errors and functional verification with temporal properties. Despite recent advances in model checking, limitations on the complexity could jeopardize the verification of complex systems. In such cases, simulations and tests are conducted to achieve greater coverage and stress of the system, and of course to provide valuable information about its behavior. As results are presented: the development and verification of an Instruction set Simulator for the MSP430 microcontroller; two hybrid verification scenarios of a fuel injection control system; a virtual platform simulation of a mechanical control system, considering physical models of the process; and finally, the specification, implementation, and testing of an onboard computer of a CubeSat, a rather complex system consisting of three processing units and a real-time operating system. These results serve as a demonstration of the potential of the methodology and demonstrate the importance of verification in the early stages of design.

Keywords: *Model Checking*, Virtual Platforms, Embedded Software, Early Verification.

LISTA DE ILUSTRAÇÕES

Figura 1 – Relação entre erros introduzidos, detectados e custo de correção. 25
Figura 2 – Potencial de aceleração no processo de desenvolvimento, verificação e depuração através de Plataformas Virtuais (VPs). 26
Figura 3 – Organização da dissertação. 29
Figura 4 – Procedimento generalizado para teste de *software*. 38
Figura 5 – Visão estrutural simplificada de *Model Checking*. 43
Figura 6 – Visão estrutural do ESBMC. 46
Figura 7 – Visão esquemática das três principais fases da metodologia utilizada com suas principais atividades. 59
Figura 8 – Fluxograma representando a relação entre as fases da metodologia utilizada[1]. 62
Figura 9 – Fluxograma representando as etapas internas da Fase 1. . . 64
Figura 10 – Fluxograma representando as etapas internas da implementação de verificação de modelos com ArchC. 66
Figura 11 – Detalhes da Fase 2, desenvolvimento e verificação incremental. 69
Figura 12 – Máquinas de estados que indicam os valores fornecidos pelos sensores (normal, falha). 75
Figura 13 – Máquina de estados que indica o número de sensores em falha. 75
Figura 14 – Máquina de estados que controla os modos de operação do sistema. 76
Figura 15 – Diagrama esquemático mostrando os módulos do FloripaSAT. 79
Figura 16 – Diagrama da Plataforma Virtual do FloripaSAT. 79
Figura 17 – Estados de funcionamento do *On Board Data Handling* (OBDH). 81
Figura 18 – Diagrama de sequência para as tarefas realizadas no estado S1. 82
Figura 19 – Diagrama de sequência para as tarefas realizadas no estado S2. 83
Figura 20 – Diagrama de sequência para as tarefas realizadas no estado S3. 84
Figura 21 – Grafo Estático de Chamada de Função para o SCIC. 95
Figura 22 – *Testbench* montado para simulação do SCAE. 100
Figura 23 – Comparação entre as duas simulações realizadas em *Model Based Design* (MBD) e *Electronic System Level Design* (ESL). . 101
Figura 24 – Diferença absoluta entre os sinais de controle obtidos com MBD e ESL. 101
Figura 25 – Visão Estrutural de SystemC. 115

Figura 26 – Classificação Típica de ISSs Desenvolvidos em SystemC. . . 118
Figura 27 – Comparação entre as Abstrações de Modelagem de ISS. . . 119
Figura 28 – Simbologia utilizada para os fluxogramas criados. 125

LISTA DE TABELAS

Tabela 1 – Operadores temporais em Lógica Temporal Linear (LTL). . 41
Tabela 2 – Ganhadores em cada categoria da competição SV-COMP 2015. 49
Tabela 3 – Sumário dos resultados obtidos. Número total de instruções executadas (Σinst), número de vezes que os monitores foram executados (ΣP). 92
Tabela 4 – Resultados obtidos das funções Fase 2, procura por erros de implementação. 95
Tabela 5 – Propriedades especificadas para verificação funcional do SCIC. 97
Tabela 6 – Verificação das propriedades (Tabela 5) com ESBMC 98
Tabela 7 – Resultados obtidos durante simulação do SCIC, considerando monitores para propriedades 5 a 11. 99
Tabela 8 – Propriedades especificadas para verificação funcional do computador de bordo. 103

LISTA DE ABREVIATURAS E SIGLAS

ART	*Adaptive Random Testing*.
ASIC	*Application-Specific Integrated Circuit*.
BDD	*Binary Decision Diagram*.
BMC	*Bounded Model Checking*.
DSE	*Dynamic Symbolic Execution*.
EPS	*Electrical Power System*.
ESBMC	*Efficient SMT-Based Context-Bounded Model Checker*.
ESL	*Electronic System Level Design*.
ETR	Estação Terrestre.
FDA	*Food and Drug Administration*.
FPGA	*Field Programmable Gate Arrays*.
GSCF	Grafo Estático de Chamada de Função.
ISR	*Interrupt Service Routine*.
LDA	Linguagem para Descrição de Arquitetura.
LTL	Lógica Temporal Linear.
MBD	*Model Based Design*.
OBDH	*On Board Data Handling*.
PI	Propriedade Intelectual.
PIL	*Processor-in-the-Loop*.
PSL	*Property Specification Language*.
SAT	*Satisfatibilidade* Booleana.
SCAE	Sistema de Controle de Acelerador Eletrônico.
SCIC	Sistema de Controle de Injeção de Combustível.

SIL *Software-in-the-Loop*.
SLD *System Level Design*.
SMT *Satisfiability Modulo Theories*.
SoC *System-on-chip*.

TLM *Transaction Level Modeling*.
TTC *Telemetry, Tracking and Command*.

UML *Unified Modeling Language*.

SUMÁRIO

Sumário **21**

1 Introdução **23**
1.1 Contextualização 23
1.2 Motivação 24
1.3 Objetivos do Trabalho 27
1.3.1 Objetivos Específicos 28
1.4 Organização da Dissertação 28

2 Conceitos e Definições **31**
2.1 Metodologias para Desenvolvimento de Sistemas 31
2.1.1 Metodologia MBD 32
2.1.2 Metodologia ESL 33
2.2 Plataformas Virtuais para Desenvolvimento e Verificação de *Software* 35
2.2.1 Simulação de Sistemas de *Software* 35
2.2.2 Plataformas Virtuais com ArchC 36
2.3 Verificação de *software* embarcado: *Teste e Verificação Formal* . . . 37
2.3.1 Teste de *software* 38
2.3.2 Verificação Formal 40
2.3.2.1 Especificação de Sistemas: LTL 40
2.3.2.2 *Model Checking* 42
2.3.2.3 *Bounded Model Checking* 44
2.3.2.4 *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) . . 45
2.4 Conclusões e Discussão 46

3 Trabalhos Relacionados **47**
3.1 Verificação Formal: *Model Checking* 47
3.1.1 Atual Panorama de *Model Checking* 47
3.1.2 Verificação de Propriedades Temporais Usando o ESBMC 48
3.2 Verificação por Simulação 50
3.2.1 Geração Automática de Testes 51
3.2.1.1 Execução Simbólica 51
3.2.1.2 Testes Randômicos 52
3.2.2 Estratégias de Testes 53
3.2.2.1 Testes Combinatórios 53
3.2.2.2 Testes Baseados em Modelos 54
3.2.3 Desafios Futuros em Teste de *Software* 54
3.3 Métodos Híbridos 56

3.4 Conclusões e Discussão 58

4 Metodologia Proposta 59
4.1 Descrição Geral da Metodologia 59
4.2 Relação entre as Fases da Metodologia Utilizada 61
4.3 Descrição Detalhada da Fase 1 63
4.4 Descrição Detalhada da Fase 2 65
4.5 Descrição Detalhada da Fase 3 70
4.6 Conclusões e Discussão 71

5 Implementações e Resultados 73
5.1 Descrição dos Estudos de Caso 73
5.1.1 Sistema de Controle de Injeção de Combustível (SCIC) 73
5.1.2 Sistema de Controle de Acelerador Eletrônico (SCAE) 77
5.1.3 Computador de Bordo de um *CubeSat* 78
5.1.3.1 Conceitos Gerais *CubeSats*: *Projeto FloripaSAT* 78
5.1.3.2 Descrição e Especificação do Estudo de Caso: *Computador de bordo FloripaSAT* 80
5.2 Resultados 85
5.2.1 Modelagem do Microcontrolador MSP430 com ArchC 85
5.2.1.1 Verificação de Modelos Gerados pelo ArchC 88
5.2.2 Verificação do *SCIC* 92
5.2.2.1 Criação da Plataforma para Simulação 92
5.2.2.2 Procura de Erros de Implementação com o ESBMC 94
5.2.2.3 Verificação de Propriedades Temporais com ESBMC 95
5.2.2.4 Simulação e Monitoramento das Propriedades 98
5.2.3 Plataforma Virtual para o SCAE 99
5.2.4 Simulação e Verificação do Computador de Bordo do *CubeSat* . 101
5.3 Conclusões e Discussão 104

6 Conclusões e Trabalhos Futuros 105
6.1 Contribuições 105
6.2 Trabalhos Futuros 106

Referências 107

APÊNDICE A Modelagem de Sistemas com SystemC 115

APÊNDICE B Programa para geração dos valores randômicos . 123

APÊNDICE C Simbologia para os Fluxogramas Utilizados . . . 125

APÊNDICE D Publicações 127

1 INTRODUÇÃO

1.1 Contextualização

A constante evolução tecnológica da nossa sociedade tem aumentado significativamente a inserção dispositivos eletrônicos em nosso cotidiano. Essa afirmação pode ser confirmada analisando, por exemplo, o contexto de computação pervasiva. Mark Weiser cunhou o termo *Ubiquitous Computing* menos de 30 anos atrás. Desde então, dispositivos eletrônicos estão tão inseridos em nossas rotinas que o simples fato de mencionar computação pervasiva pode parecer obsoleto. Analisando por outro ângulo, o número de dispositivos móveis conectados é estimado para 12 bilhões[1] até 2020. O crescimento médio anual nesse período seria de 30%, em números atuais. Embora esses dados simbolizem o ponto de vista de telefonia móvel, a área de sistemas embarcados em geral apresenta propensões similares. Na área automotiva, por exemplo, não é incomum a utilização de centenas de unidades de processamento, principalmente em veículos *high-end* (GEORGAKOS; SCHLICHTMANN; SCHNEIDER, 2013).

Associada a essa perspectiva, há constante preocupação com segurança e integridade. Como apontado por Council, Jackson e Thomas (2007), a automação tende a reduzir a probabilidade de falha; no entanto, a sua severidade é muitas vezes agravada. Um exemplo catastrófico (resultou em perda de vidas humanas) dessa afirmação é exposto por Jackson (2004). Durante a guerra do Afeganistão (início dos anos 2000), um soldado que estava para iniciar um ataque aéreo coordenado via GPS, percebeu que o dispositivo que estava usando para fornecer as coordenadas, estava com nível de bateria baixo. Após substituir a bateria iniciou o ataque, sem saber que o dispositivo havia sido programado para inicializar com as coordenadas apontando para si mesmo. Esse exemplo, embora catastrófico, revela perspectivas (tais como interação entre *hardware*, *software* e com usuário) consideravelmente difíceis de serem exploradas em tempo de projeto. A tendência, perante o processo de aumento na produção e inserção em diversas áreas, é que o número de situações a serem previstas durante verificação seja maior, tornando cada vez mais difícil garantir integridade funcional.

Todos esses fatores associados à complexidade crescente, principalmente do ponto de vista de *software*, impõem dificuldades ainda maiores, tanto para desenvolvimento quanto para verificação. Na área aeroespacial, por exemplo, o crescimento no volume de *software*, considerando dois aviões comerciais (Airbus A330 para o A380) produzidos em um período de 13 anos de dife-

[1] Segundo Cisco (2015).

rença, chega a 5 vezes[2] (WIELS et al., 2012). Mesmo assim, frente tamanha complexidade, o único incidente reportado no modelo mais recente (A380), foi posteriormente relacionado a uma falha mecânica (NASA, 2015). Deve-se levar em conta que a área aeroespacial foi alvo de grandes avanços, não somente em termos tecnológicos e complexidade, mas em padrões e certificações para segurança funcional. Em outras áreas, como equipamentos médicos, apesar de uma falha não fornecer ameaças para um grande número vidas, uma única falha pode ser letal. Apesar disso, bases de dados que registram incidentes envolvendo dispositivos médicos, mostram surpreendentes 30.000 mortes (no período 1985 a 2005) envolvendo falhas (COUNCIL; JACKSON; THOMAS, 2007). Evidente que uma grande percentagem desse valor não está diretamente relacionado a falhas causadas por um defeito em *software*. Porém, o relatório publicado pelo *Food and Drug Administration* (FDA), aponta que as causas mais frequentes de *recalls*, de dispositivos médicos, estão relacionadas a problemas de projeto e *software* (FDA, 2012). Além disso, iniciativas para resolver esse problema poderiam reduzir cerca de 400 *recalls* por ano.

Em suma, por um lado, sistemas extremamente complexos (área aeroespacial) e difíceis de serem desenvolvidos e verificados, apresentam baixo índice de falhas relacionadas com *software*; por outro lado, dispositivos que não são tão complexos (área médica), em comparação, mas que apresentam número de falhas elevadas, com considerável porção sendo identificada com origem em *software*. Mesmo considerando as diferentes áreas e proporções de produção dos produtos, a preocupação com esse contraste tem sido abordada recentemente em diversas normas e certificações, justamente numa tentativa de redução de falhas e garantia de segurança funcional.

Questões importantes de serem observadas frente a conjuntura descrita nos parágrafos anteriores, são as razões que levam ou que causam essas falhas do ponto de vista de *software*. Na realidade, dificilmente os fatores possam ser determinados e discriminados de forma objetiva. Alguns indicativos, no entanto, servem de motivação para este trabalho. Talvez a forma mais correta de abordar o problema é analisar inicialmente o que o mercado exige e quais são as alternativas técnicas, levando em conta ferramentas e tendências atuais.

1.2 Motivação

Recentes pesquisas indicam (ver UBM Tech (2015)) que a relação em recursos totais investidos no desenvolvimento de sistemas embarcados (considerando tempo, investimento e mão de obra), é cerca de 60% em *software* e 40% em *hardware*. Esse valor pode ser o primeiro indicativo das necessida-

[2] De 20 Milhões para 100 Milhões de linhas de código.

des em soluções, dentro do contexto descrito anteriormente. Além disso, o mercado exige alta produtividade com o menor *time-to-market* possível. Tal exigência agrava a situação, pois para certa empresa ter chance de sucesso, deve empenhar grande parte do tempo no desenvolvimento de *software*, o qual está se tornando cada vez mais complexo e difícil de verificar. Tudo isso em um período de tempo cada vez menor, devido a competitividade e pressão do mercado. Essas características não técnicas podem ser um dos fatores determinantes em situações tão opostas, como descrito anteriormente.

Uma solução lógica ao problema seria uma metodologia de desenvolvimento, centrada em *software* (afinal *software* tende a tomar grande parte dos recursos de projeto), que forneça mecanismos para verificação já em fases iniciais. Essa solução atende idealmente ao problema identificado e ainda poderia ter o potencial de acelerar o processo de desenvolvimento de *software*.

A Figura 1 mostra a quantidade de erros introduzidos e detectados, considerando etapas típicas do projeto de *software* e o custo de correção. Como não há formas práticas de eliminar a quantidade de erros introduzidos, o ideal seria, visando redução de custos e tempo total de projeto, aproximar as curvas azul e preta (erros introduzidos e detectados) o máximo possível.

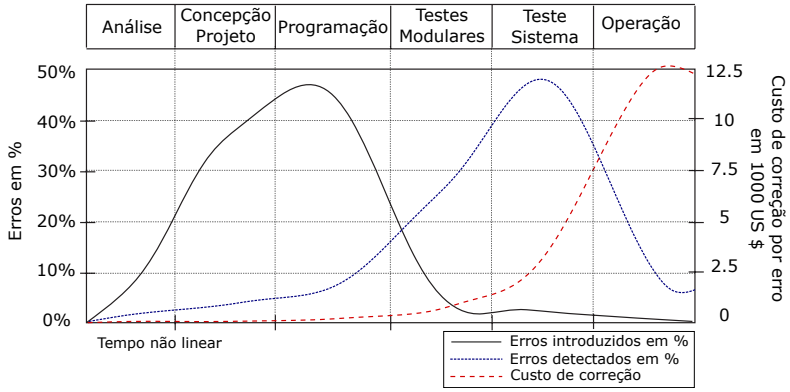

Figura 1 – Relação entre erros introduzidos, detectados e custo de correção. Adaptado de Baier e Katoen (2008).

Seguindo fluxo de projeto tradicionais, em que *hardware* e *software* são desenvolvidos separadamente, sendo integrados somente em fases posteriores de projeto, é relativamente complicado aproximar tais curvas. Uma das razões é que as etapas de teste não podem ser adiantadas, pois pode ser que o *hardware* ainda não esteja pronto para testes. Outro problema, talvez mais grave, é que decisões quanto a arquitetura de *hardware* devem ser realizadas durante a fase de concepção, etapa em que não há tanta liberdade para exploração do espaço

de projeto.

Diante dessa situação algumas soluções são propostas na literatura, fornecendo alternativas para esse fluxo. Uma possível solução, que aparenta ter certo sucesso e aumento em uso recentemente, é utilizar linguagens de modelagem que fornecem suporte para *hardware* e *software*, e com base nisso criar plataformas virtuais do sistema (AARNO; ENGBLOM, 2014). Essa abordagem é particularmente interessante, pois fornece os meios para explorar diversas arquiteturas possíveis, sem necessidade de experimentações em *hardware* (o que pode ser custoso em tempo). Além disso, podem ser utilizadas como base para desenvolvimento, verificação e depuração do *software* embarcado. Na Figura 2, é representado qualitativamente o potencial que a utilização de plataformas virtuais pode inserir no processo de desenvolvimento. Inicialmente, cria-se a plataforma virtual (região em verde); com isso, o desenvolvimento e verificação de *software* pode iniciar muito antes, reduzindo o tempo total de projeto (AARNO; ENGBLOM, 2014).

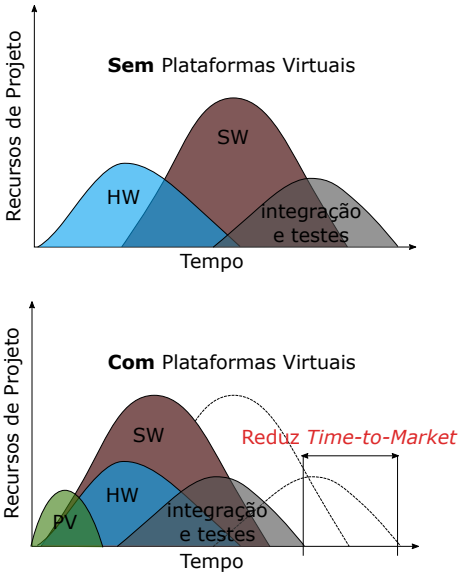

Figura 2 – Potencial de aceleração no processo de desenvolvimento, verificação e depuração através de Plataformas Virtuais (VPs). Adaptado de Aarno e Engblom (2014).

Claramente, o potencial dessa abordagem harmoniza com a situação atual do mercado de sistemas embarcados, com as tendência em complexidade, além de ter o potencial de poder ser utilizada no contexto idealizado de

aproximação das curvas na Figura 1.

Do ponto de vista de verificação, há diversos métodos e formas de garantir integridade funcional de *software*. O mais comum na indústria é realizar testes, ou seja, executar o *software* em diversos cenários até determinado critério ser alcançado (e.g., cobertura de código). Essa abordagem, embora não completa e basicamente manual, é a mais utilizada. Análises estáticas, sem execução, também podem ser aplicadas e apesar de certas limitações, fornecem garantias quanto a integridade. Em termos práticos, é comum a aplicação de abordagens que combinem ambas as metodologias (D'SILVA; KROENING; WEISSENBACHER, 2008). Apesar das limitações individuais, quando combinadas, tendem a aumentar suas capacidades em verificação. Dessa forma, em um contexto atual, não é coerente abordar verificação sem considerar ambas as metodologias. Verificar as mesmas porções do sistema com ferramentas diferentes pode não ser uma má ideia, mas geralmente em fases inicias de projeto há interesse em verificar o máximo possível. Assim, é importante extrair o máximo das ferramentas de análises estáticas e sua completude (em certos aspectos) e posteriormente, utilizar testes como forma de depuração e auxílio, quando as ferramentas de análise estáticas chegam ao seu limite.

Devido as recentes tendências e pesquisas em *Model Checking* qualquer metodologia de verificação que pretenda abordar o estado da arte deve levar em conta os avanços nessa área, o mesmo vale para plataformas virtuais.

Este trabalho considera todos esses aspectos relacionados até aqui, para propor uma metodologia antecipada de desenvolvimento voltado para verificação de *software* embarcado. Essa metodologia pode ser utilizada já nas primeiras fases de projeto, utilizando plataformas virtuais para desenvolvimento. O potencial dessa metodologia tende a favorecer a construção antecipada de sistemas íntegros funcionalmente, atendendo assim, os requisitos de mercado.

1.3 Objetivos do Trabalho

O objetivo principal deste trabalho é descrever a metodologia de verificação e avaliar diversos cenários de aplicação considerando sistemas de complexidade e funcionalidades variadas. Essa metodologia é dividida em três fases principais ((1) Desenvolvimento da Plataforma Virtual, (2) Integração do *Software* Embarcado com a Plataforma Virtual e Verificação Incremental e (3) Verificação do Sistema todo através de Simulações) e pode ser inserida dentro de qualquer fluxo de desenvolvimento e verificação.

1.3.1 Objetivos Específicos

Os objetivos específicos são relacionados como *Milestones* da dissertação. Seguindo ordem de importância são eles:

i. Propor a metodologia para verificação antecipada de *software* embarcado;

ii. Aplicação da metodologia proposta a um sistema complexo de um *CubeSat*;

iii. Observação de propriedades temporais considerando análises dinâmicas e estáticas;

iv. Criação e verificação de um modelo para o microcontrolador MSP430, utilizando a linguagem de descrição de arquiteturas ArchC;

v. Analisar cenários complexos de verificação funcional aplicando a metodologia proposta;

vi. Integração das metodologias *Model Based Design* e *Electronic System Level Design*;

1.4 Organização da Dissertação

O restante deste trabalho está organizado conforme representado na Figura 3. A discussão é iniciada na introdução, de forma mais geral, sendo limitada progressivamente para o tema central deste trabalho. Na conclusão, o processo inverso é realizado.

Figura 3 – Organização da dissertação. O formato das figuras indica como as informações são apresentadas.

2 CONCEITOS E DEFINIÇÕES

Neste capítulo, são revistos os conceitos e definições usados ao longo desse texto.

Na primeira parte deste capítulo, são apresentadas metodologias típicas de desenvolvimento de sistemas embarcados. O ponto de vista é mais voltado para verificação de *software* e como as metodologias de desenvolvimento de sistemas apresentadas podem contribuir de forma eficiente para o processo de verificação funcional. Na segunda parte, são apresentados conceitos gerais sobre especificação e verificação de sistemas de *software*. Finalmente, esses conceitos são discutidos dentro do panorama de trabalho na Seção 2.4.

2.1 Metodologias para Desenvolvimento de Sistemas

No desenvolvimento de sistemas embarcados, a necessidade da utilização de metodologias de projeto foi crescendo gradativamente com a complexidade com que tais sistemas foram adquirindo ao longo do tempo. Para lidar com essa necessidade e com requisitos mais rigorosos, o paradigma de projeto de tais sistemas mudou de forma considerável. Desde um cenário em que grande parte dos projetos eram desenvolvidos manualmente, para um cenário altamente automatizado e abstrato que se tem hoje. Contudo, o conceito fundamental não mudou, somente tomou novas dimensões conforme complexidade, recursos e esforço no desenvolvimento. Possivelmente uma consequência das abstrações de projeto, introduzidas para lidar com a complexidade dos sistemas.

A divisão entre funcionalidades implementadas em *software* e *hardware* sempre foi uma problemática do projeto de sistemas embarcados. Nas últimas décadas, no entanto, avanços no desenvolvimento de metodologias de projeto introduziram abstrações para realização do projeto simultâneo de *hardware* e *software*. Novas metodologias e ferramentas surgiram nesse contexto, como *Electronic System Level Design* (ESL) e SystemC[1] TLM. Por outro lado, metodologias já existentes para desenvolvimento de algoritmos para controle e processamento de sinais como (*Model Based Design* (MBD)), adicionaram funcionalidades aos seus fluxos de desenvolvimento de forma a abranger também o projeto integrado de tais sistemas. Essa tendência é facilmente observável nas funcionalidades de ferramentas como Simulink® e LabView®.

No cenário atual, essas duas metodologias, apesar de serem baseadas em conceitos e implementações completamente diferentes, podem ser utilizadas dentro de um mesmo fluxo de desenvolvimento. É importante destacar

[1] Uma revisão de SystemC e *Transaction Level Modeling* (TLM) é realizada no Apêndice A

que a combinação trás muitas vantagens, pois pode unir os domínios sem comprometimento, não somente para desenvolver um sistema, mas também para garantir qualidade e funcionalidade através de verificação. Portanto, não faz sentido uma revisão que não aborde tais metodologias, suas vantagens e desvantagens, mesmo que o objetivo seja somente o processo de verificação de *software*.

2.1.1 Metodologia MBD

Model Based Design é uma metodologia largamente utilizada que surgiu inicialmente com o propósito de facilitar o desenvolvimento de sistemas de controle, mas foi adaptada para atender a diversos outros fluxos de desenvolvimento. Nessa abordagem, um novo elemento é introduzido entre a especificação do sistema e o código final, o *modelo*[2], origem da denominação MBD. No fluxo de desenvolvimento, o *modelo* captura a funcionalidade do sistema e é desenvolvido em um ambiente gráfico sendo posteriormente traduzido para uma implementação específica (CONRAD; MOSTERMAN, 2013).

Simulink® é a ferramenta com maior popularidade e funcionalidades em se tratando de MBD (MathWorks, 2016). Os *modelos* desenvolvidos em Simulink® são baseados em diagramas de blocos. Cada bloco pode representar uma operação matemática, um elemento dinâmico ou um subsistema que pode ser decomposto em outro diagrama de blocos. As arestas de um diagrama representam as relações entre entradas e saídas de cada bloco. As descrições criadas em Simulink® podem ser puramente declarativas, imperativas ou ambas. Descrições declarativas são representadas pelo uso de blocos operacionais preestabelecidos, como integradores, somadores ou blocos dinâmicos. Os elementos imperativos são introduzidos diretamente através de blocos Stateflow® (implementam máquinas de estado finitos) ou utilizando funções escritas em Matlab®.

Do ponto de vista de verificação, vários trabalhos foram propostos, tanto para verificação dos *modelos* desenvolvidos em Simulink®[3] (como o trabalho proposto por Meenakshi, Bhatnagar e Roy (2006)), quanto para verificação do processo de geração automática do *software* (STAATS; HEIMDAHL, 2008).

As principais vantagens de MBD residem justamente em explorar, ainda em fases iniciais, o espaço de projeto através de *modelos* abstratos, aumentando, assim, a produtividade em até 50% (CONRAD; MOSTERMAN, 2013). As

[2] O significado de modelo é destacado aqui pois refere-se a um tipo específico: aquele desenvolvido em ambiente gráfico em um fluxo MBD.

[3] Até mesmo uma ferramenta comercial da própria MathWorks® (MATHWORKS, 2015).

simulações desenvolvidas em Simulink®, por exemplo, podem incorporar diversos elementos heterogêneos como *Processor-in-the-Loop* (PIL) ou *Software-in-the-Loop* (SIL). Apesar disso, outras abordagens de desenvolvimento se destacam com relação a MBD, principalmente na integração e modelagem de elementos de *software* e *hardware*. Porém, como MBD é uma das abordagens mais populares, alguns trabalhos propõem cossimulação ou tradução dos modelos em outras formas de representação para posterior integração com outras metodologias de desenvolvimento (BECKER; KUZNIK; MUELLER, 2014).

2.1.2 Metodologia ESL

O termo ESL surgiu em meados dos anos 90 como um substituto para certos termos que já estavam em desuso, como *System Level Design* (SLD) entre outros (MARTIN; BAILEY; PIZIALI, 2010). Rapidamente, a comunidade adotou o termo para denotar o projeto simultâneo de *hardware* e *software*[4]. No entanto, conforme a metodologia foi amadurecendo e expandindo sua aplicabilidade, definições mais completas foram surgindo. Uma das definições mais amplas, que atende o que realmente é ESL em um cenário atual, é apresentada por Martin, Bailey e Piziali (2010). Em sua definição, ESL remete ao uso das abstrações e ferramentas necessárias para aumentar o entendimento sobre determinado sistema, aumentando a probabilidade de sucesso na implementação, sempre levando em conta os requerimentos de projeto e limite de orçamento. Na realidade ESL compreende uma miscelânea de metodologias, tanto para desenvolvimento de produtos *board-level*, *System-on-chip* (SoC) e também *Field Programmable Gate Arrays* (FPGA).

Um dos principais aspectos que caracterizam a metologia ESL é o desenvolvimento de uma especificação executável; ou seja, modelos geralmente implementados em uma linguagem, como por exemplo Rosetta (vide Alexander (2011)) ou SystemC (ver padrão 1666 IEEE (2011)). A linguagem que melhor representa ESL, em se tratando de sistemas, é SystemC, que é um conjunto de bibliotecas[5] de classes e macros construídas sobre a linguagem C++ (RIGO; AZEVEDO; SANTOS, 2011). Rosetta, por outro lado, possui certa relevância, porém, devido a suas características ou limitações não foi muito utilizada pela indústria, embora esteja a caminho de se tornar o padrão IEEE™ 1699 (ver IEEE (2008)), o que poderia aumentar sua adoção pela indústria.

[4] É importante perceber que ESL é um termo geral que engloba outras metodologias como HW/SW *Co-design*, ou até mesmo *Platform Based Design*.

[5] Apesar de não se tratar de uma linguagem por si só, é comum na literatura se referir a SystemC como uma linguagem.

Especificação executável pode ser vista como um modelo funcional, o qual pode ser implementado em vários níveis de abstração diferentes. Normalmente, em fases iniciais de projeto, um modelo simplificado é utilizado para auxiliar no processo de levantamento de requisitos e especificações do sistema. Conforme o projeto vai se encaminhando para sua fase final, esse modelo vai sendo refinado e acaba se tornando uma referência para implementações em *hardware*; ou ainda, através das ferramentas apropriadas pode ser sintetizado em um fluxo FPGA, ou *Application-Specific Integrated Circuit* (ASIC), por exemplo.

Uma das tendências dentro da metodologia ESL é de utilização de plataformas virtuais para auxiliar e acelerar a produção do produto final. Do ponto de vista de *software*, em fluxos convencionais[6] de desenvolvimento, somente após a definição e criação de um protótipo em *hardware* é que se iniciaria as implementações do *software*. Com a utilização de plataformas virtuais, no entanto, pode-se iniciar o desenvolvimento muito antes, com total liberdade para experimentação e exploração do espaço de projeto. Essa abordagem vem sendo largamente utilizada pela indústria com diversas ferramentas comerciais como: Synopsys Virtualizer®, Cadence Virtual System Platform (VSP)® e Imperas Instruction Set Simulator®.

É comum à maioria dessas plataformas virtuais a utilização certos blocos construtivos, principalmente na descrição de uma arquitetura, como processadores, memórias, elementos não programáveis e barramentos de comunicação (RIGO; AZEVEDO; SANTOS, 2011). Nesse contexto, emuladores[7] de conjuntos de instruções geralmente estão no centro das plataformas virtuais como um ou mais núcleos de processamento. A criação desses simuladores pode ser uma tarefa árdua, dependendo da complexidade do conjunto de instruções em questão e do nível de abstração necessário. Apesar disso, algumas técnicas permitem certa padronização do modelo, facilitando o processo de geração automática através de descrições abstratas. Partindo desse princípio, é possível, através de uma linguagem abstrata, automatizar parte do processo e gerar uma implementação em uma outra linguagem de modelagem, como SystemC. As linguagens abstratas utilizadas na descrição da arquitetura em questão são denominadas Linguagens para Descrição de Arquiteturas (LDAs).

Muitas LDAs foram propostas na literatura; uma comparação mais detalhada entre algumas dessas linguagens está fora do escopo desse trabalho. Vale a pena destacar que ArchC oferece certa versatilidade para modelagem

[6] Fluxos em que *hardware* e *software* são desenvolvidos separadamente e integrados em fases posteriores.

[7] Também denominados *Instruction Set Simulators* ou simuladores de conjunto de instruções.

de plataformas virtuais dentro do contexto ESL e SystemC; por esse e outros motivos, é utilizada nesse trabalho.

Algumas LDAs que fornecem implementações e conexões com SystemC permitem a criação de plataformas virtuais para o sistema, incluindo diversos periféricos e unidades de processamento. Essa abordagem vem sendo explorada extensivamente por ferramentas comerciais. Um fator comum é a utilização da linguagem SystemC e do padrão *Transaction Level Modeling* (TLM)[8] para criação dessas plataformas. SystemC e TLM trazem mais vantagens para criação de plataformas virtuais, devido, principalmente, à abstração da comunicação entre componentes através da modelagem de transações e de facilidades para modelagem de elementos de *hardware* e *software* em SystemC.

É importante ressaltar, dentro dessa perspectiva de automatização da criação do simulador utilizando SystemC e TLM, os fluxos de trabalho e métodos que são escaláveis a partir de simuladores para sistemas e plataformas virtuais, como proposto pela LDA ArchC (CARDOSO, 2015).

Essa abordagem voltada para criação de plataformas virtuais, iniciando com a criação do simulador para a arquitetura alvo, é uma das facetas da metologia ESL. Isso não impede que o *software* embarcado final, como por exemplo um algoritmo de controle, seja desenvolvido usando MBD e posteriormente, integrado com à plataforma virtual para depuração, verificação e simulação a nível de sistema. A integração entre essas diferentes abordagens une o melhor dos dois domínios e traz vantagens evidentes tratando-se de sistemas.

2.2 Plataformas Virtuais para Desenvolvimento e Verificação de *Software*

Nas seções anteriores foi exposto brevemente algumas metodologias para desenvolvimento de sistemas. Nessa seção, são apresentados alguns conceitos relacionados ao processo de elaboração de plataformas virtuais dentro do contexto ESL.

2.2.1 Simulação de Sistemas de *Software*

Um fator comum, independente da metodologia ou metodologias utilizadas, é a necessidade de se verificar determinado modelo de forma a assegurar sua corretude. Em se tratando de sistemas complexos, algum tipo de simulação do sistema como um todo geralmente é utilizada. É claro que, nesses casos, é necessário algum mecanismo para simulação e modelagem do *software* em questão.

[8] Um padrão para modelagem de comunicação entre componentes de *hardware* em forma de transações que visa principalmente a interoperabilidade entre modelos.

Como apontado por (SMITH; NAIR, 2005) há duas abordagens típicas para simulação de sistemas de *software*. A primeira, mais simples e muito utilizada, é criar um interpretador para o conjunto de instruções que busca, decodifica e executa cada instrução em *software*. A segunda, seria traduzir de forma estática ou dinâmica o código binário para uma outro conjunto de instruções suportados por *hardware* (na máquina hospedeira). Como na segunda opção as instruções são executadas diretamente em *hardware* o ganho de desempenho, mesmo considerando *overhead* de tradução, pode ser considerável, podendo alcançar 2 à 3 ordens de grandezas (BARTHOLOMEU, 2005).

Os simuladores comerciais modernos que apresentam bom desempenho geralmente utilizam alguma técnica de tradução binária ou de otimização do interpretador para reduzir o *overhead* de emulação. Há diversos trabalhos que seguem esse caminho, nesse contexto deve-se destacar os trabalhos que exploram alguma técnica para combinação entre metodologias de desenvolvimento e verificação (BECKER; KUZNIK; MUELLER, 2014).

A abordagem para criação de um simulador utilizando SystemC e TLM geralmente se baseia na criação de um interpretador. Por um lado, tem-se um ganho considerável para modelagem do sistema; porém, o desempenho fica comprometido em comparação com tradução binária. Todavia, há certas otimizações que aumentam o desempenho do interpretador e já são incorporadas em algumas metodologias, como apresentado por Bartholomeu (2005). Levando em conta um certo compromisso, pode-se conseguir resultados relevantes, sem abrir mão de desempenho e facilidade de modelagem. Essa é uma das razões pelo qual é demonstrado a utilização de ArchC, no contexto de desenvolvimento de verificação de *software* embarcado.

2.2.2 Plataformas Virtuais com ArchC

`ArchC`[9] é uma LDA de código aberto capaz de gerar implementações em SystemC e TLM. O fluxo de trabalho para criação de um novo simulador é relativamente direto, e reflete um dos pontos fortes com relação ao ArchC. Basicamente, a partir de algumas informações gerais sobre a arquitetura, tais como tamanho da palavra, conjunto de registradores, formatos e campos de cada instrução, *endianess* e memória disponível, ArchC processa as entradas fonte e gera os arquivos que compõe o simulador. Posteriormente, é necessário implementar o comportamento de cada instrução diretamente nos protótipos gerados pela ferramenta.

Funcionalidades adicionais podem ser incorporadas no modelo à medida de necessidade. Como sugerido em (RIGO; AZEVEDO; SANTOS, 2011)

[9] <https://github.com/ArchC/ArchC>

um dos passos iniciais, após o desenvolvimento de uma versão estável do simulador, seria criar um servidor de depuração, interno ao modelo, para a depuração do *software* que está sendo executado. Para melhor testar e validar o modelo Rigo, Azevedo e Santos (2011) também propõem um conjunto de *benchmarks* para serem executados e depurados através dessa interface de depuração, facilitando assim, a detecção de possíveis erros na modelagem. Outras funcionalidades a serem destacadas, que podem ser inseridas no simulador através do ArchC: i) emulação de chamadas para o sistema operacional (*syscall*); ii) redirecionamento de ferramentas binárias para a arquitetura modelada; iii) integração com blocos de Propriedade Intelectual (PI) desenvolvidos em SystemC; iv) simulação de sistemas operacionais completos, como mostrado por Zijlstra (2015) e Cardoso (2015).

A partir da modelagem do simulador, ArchC permite criar plataformas virtuais sem preocupação com *overhead* de implementação. Os modelos são desenvolvidos de forma modular, o que permite a criação de várias instâncias do mesmo modelo, nesse caso um módulo em SystemC (o modelo em si). Um exemplo de plataforma virtual para um *CubeSat*,[10] desenvolvido totalmente em SystemC usando ArchC, é apresentado por Zijlstra (2015).[11]

Para gerenciamento dessas plataformas, uma série de ferramentas e *scripts* são distribuídos juntamente com o ArchC. A destacar o *ArchC Reference Platform* (ARP) que possui funcionalidades para armazenamento e distribuição, bem como gerenciamento da plataforma como um todo (AZEVEDO; ALBERTINI; RIGO, 2010).

2.3 Verificação de *software* embarcado: *Teste e Verificação Formal*

Um fato bem conhecido com relação ao processo de verificação de sistemas em geral (não somente sistemas que são baseados em *software*), é que este pode consumir grande parte do tempo de projeto (UBM Tech, 2015). Normalmente, estima-se esse valor em torno de 50%. Com base nisso, já na década de 70, com a corrida espacial e crescimento considerável na complexidade dos sistemas embarcados críticos, encontra-se publicações e estudos com maior preocupação em assegurar qualidade dos sistemas (DELAMARO; MALDONADO; JINO, 2007).

Atualmente, as alternativas encontradas para verificação de *software* podem ser divididas em duas categorias: análises estáticas, não requerem

[10] *CubeSat* é um formato de satélite para pesquisa em que as dimensões e custos são reduzidos, tipicamente realizado em uma estrutura cúbica com 10 cm de aresta.

[11] Esse trabalho foi desenvolvido no Grupo de Sistemas Embarcados (GSE), na Universidade Federal de Santa Catarina (UFSC), e é utilizado em partes desse trabalho.

execução do *software*; testes dinâmicos, o *software* é executado para poder avaliar o seu correto funcionamento. Essa classificação reflete aos métodos específicos dentro de um contexto. Em um fluxo completo de verificação, no entanto, é comum a utilização de muitos métodos diferentes para obter garantias melhores quanto a conformidade do produto em questão.

Nas seções seguintes, será apresentado um panorama geral de teste de *software* do ponto de vista funcional, análises dinâmicas. Além de uma discussão sobre métodos algorítmicos de análises estáticas, com foco principal em *model checking*.

2.3.1 Teste de *software*

O procedimento generalizado de teste de *software* é resumido de forma gráfica na Figura 4. Seguindo as definições apresentadas por Delamaro, Maldonado e Jino (2007), *domínio* de programa $D(P)$ é o conjunto dos valores de entrada possíveis para executar determinado programa P. Uma rotina de teste consiste em definir o conjunto de casos de testes T, executar o programa P com base nesse conjunto e, por fim, comparar o resultado obtido com uma especificação do programa $S(P)$. Caso sejam encontradas divergências entre a especificação e os resultados encontrados durante o teste, tem-se uma falha. É importante ressaltar o significado adotado para erro, falha e defeito. Aparentemente não há significado unânime para esses três conceitos. Assim, será adotado as definições utilizadas por Delamaro, Maldonado e Jino (2007). Segundo Delamaro, Maldonado e Jino (2007) defeito é a definição incorreta de dados; erro é a manifestação de um defeito, ou seja, um estado inconsistente alcançado durante execução; falha diz respeito ao resultado esperado para determinada tarefa, um erro pode se tornar uma falha se o resultado produzido for diferente do esperado. A utilização do termo defeito, no entanto, é substituída pelo termo erro de implementação em alguns pontos deste trabalho.

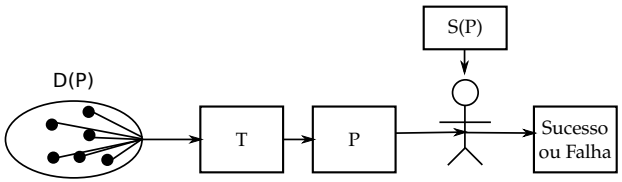

Figura 4 – Procedimento generalizado para teste de *software*. Adaptado de (DELAMARO; MALDONADO; JINO, 2007).

Três questões permanecem abertas a partir dessa descrição de uma

rotina de teste: 1) Qual o domínio D do programa P?; 2) Como definir os casos de teste T?; 3) Como são definidas as especificações $S(P)$?.

O domínio $D(P)$ é justamente o conjunto de todos os valores possíveis para o programa, considerando as permutações entre todas as variáveis e os valores que essas assumem considerando uma arquitetura. Esse conjunto pode ser claramente gigantesco, e por isso é selecionado um subconjunto de $D(P)$ para se realizar os testes, a função ou heurística que faz esse mapeamento tem como resultado o conjunto T.

Existem diversos métodos para selecionar o subconjunto T, vale a pena comentar duas heurísticas que são muito utilizadas para definição do conjunto de casos de testes em *software* e também, para testes em *hardware*. A primeira, denominada classes de equivalência, consiste em enumerar classes de entradas que devem estimular o sistema da mesma forma, gerando um comportamento observável similar para valores definidos dentro dessa classe de entrada. Por exemplo, determinando faixas de valores que se enquadram como válidos, e valores que podem ser atingidos, mas que ficam fora dessa faixa, valores inválidos. Outra heurística bastante utilizada e eficiente é considerar valores extremos, no limite das faixas de valores, levando em conta ainda classes de equivalência. Outros métodos podem ser ainda mais eficientes, mas em geral, os métodos mencionados são os mais simples e utilizados (MYERS et al., 2004).

Por fim, a definição da especificação do programa $S(P)$ pode ser feita de forma escrita, como um documento, ou através de algum formalismo, usando propriedades temporais, por exemplo. Por um lado, há necessidade de se verificar a documentação e analisar a execução do programa, para observar se há uma violação das especificações que devam ser atendidas. Por outro lado, as propriedades temporais que representam a especificação do comportamento temporal do sistema devem ser descritas, pela equipe responsável, e avaliadas por algum mecanismo durante a execução do programa. As duas abordagens salientam uma característica manual da verificação através de testes. Apesar de certas ferramentas serem capazes de realizar determinados tipos de testes de forma automática, considerando cobertura de caminhos de programas por exemplo, a aplicação em *software* embarcado pode ser limitada (ver Sen, Marinov e Agha (2005) e Cadar, Dunbar e Engler (2008)).

Em todo o projeto há fases destinadas a realização de testes para verificação e validação do *software*. Levando isso em conta, técnicas automáticas para detecção desses erros são mais interessantes por excluírem o fator humano e facilitarem a realização de testes em larga escala. Porém, muitas vezes, não é factível em tempo finito realizar testes para todos cenários possíveis. Para lidar com essa dificuldade, pode-se utilizar métodos formais, como *Model Checking*, que introduzem uma série de abstrações e realizam buscas exaustivas do

espaço de estados.

2.3.2 Verificação Formal

Originalmente os sistemas computacionais eram vistos como elementos transformacionais, em que o comportamento é representado de forma abstrata em termos de entradas/saídas. Em conjunto com o desenvolvimento de análises algorítmicas, baseadas em raciocínio matemático, uma série de conceitos e formas de representação foram elaboradas para se poder realizar afirmações ou previsões com relação ao comportamento desse tipo de sistema. No entanto, essa abstração não engloba muitos dos elementos que interagem com o ambiente, que não terminam, ou ainda que trocam informações com outros sistemas em uma rede de comunicação. O conceito de reatividade foi introduzido como abstração para essa classe de sistema. Como consequência, diversas técnicas formais foram aplicadas ou desenvolvidas especialmente para tratar esses casos (FISHER, 2011).

No âmbito de sistemas reativos, há interesse especial, dentro do tema deste trabalho, no processo de escrita de especificações e na verificação dessas especificações com relação a um dado modelo.

Nos próximos tópicos, são apresentados conceitos relacionados à lógica temporal, uma forma de representar comportamento reativo, e *Model Checking*, um conjunto de algoritmos de verificação de modelos considerando propriedades temporais.

2.3.2.1 Especificação de Sistemas: Lógica Temporal Linear (LTL)

O formalismo matemático utilizado para abstrair os argumentos em uma sequência de premissas é definido como lógica proposicional (HARRISON, 2009). Para tratar de proposições estáticas, pode-se utilizar lógica proposicional. Porém, quando há uma dependência temporal é necessário estender a base conceitual de forma a tratar o tempo, considerando um modelo apropriado.

Para lidar com essas situações dinâmicas, várias lógicas temporais foram desenvolvidas, como CTL, CTL* ou LTL. Essas lógicas utilizam os conectivos ($\wedge$, $\vee$, $\neg$, $\Rightarrow$) e adicionam operadores para representar relações temporais entre um conjunto de proposições.

Da mesma forma que do ponto de vista de lógica proposicional, as proposições devem ser mapeadas para valores **V**erdadeiros ou **F**alsos, os operadores temporais necessitam de um modelo de tempo para se poder fazer afirmações com relação as transições nesse mesmo modelo. Ou seja, no caso mais simples, o tempo é considerado como uma sequência linear (discreta) de

estados proposicionais (FISHER, 2011). Esse caso específico é denominado *Linear Temporal Logic* ou Lógica Temporal Linear (LTL).[12]

Em LTL há 5 operadores temporais, a sintaxe é resumida na Tabela 1.

Tabela 1 – Operadores temporais em LTL. Adaptado de (FISHER, 2011).

Operador	Sintaxe	Descrição
Always	Gφ	φ é **V**erdadeiro para todos os momentos futuros
Next	Xφ	φ é **V**erdadeiro para o próximo momento futuro
Eventually	Fφ	φ é **V**erdadeiro em algum[13] momento futuro
Until	φUψ	φ continua sendo **V**erdadeiro até ψ ser **V**erdadeiro em um momento futuro
Unless	φWψ	φ continua sendo **V**erdadeiro a menos que ψ se torne verdadeiro

As diversas formas de combinar esses operadores revela padrões típicos de comportamento que são encontrados em sistemas computacionais. Esses padrões de propriedades podem ser divididos, segundo Fisher (2011), em: *Safety*, *Liveness* e *Fairness*.

Propriedades *Safety*

Quando há necessidade de se realizar asserções do tipo: "*algo ruim não deve acontecer*", se utiliza propriedades *Safety*.

O formato geral $G(\varphi)$[14] em LTL, especifica esse tipo de comportamento.

Propriedades *Liveness*

Propriedades *Liveness* são utilizadas para afirmações do tipo: "*alguma coisa boa deve ocorrer*". Seu formato geral em LTL tem a seguinte sintaxe: $F(\varphi)$.

Uma aplicação de propriedades *Liveness* é para especificação de comportamento do tipo pedido/resposta. Um dos formatos possíveis é dado por: $G(\varphi_1 \Rightarrow F\varphi_2)$, onde $\varphi_{(1,2)}$ são proposições Booleanas. Sempre que φ_1 for verdadeiro, que houver um pedido, deve haver um momento futuro em que ϕ_2 se torna verdadeiro também, uma resposta.

[12] Há também lógicas que usam um modelo em ramificações do tempo é o caso de CTL e CTL*.

[13] Considerando o presente momento também.

[14] φ é uma expressão Booleana.

Propriedades *Fairness*

A ideia de propriedades *Fairness* está associado a combinação dos operadores temporais G e F na forma $GF(\varphi)$, também conhecidos por *infinitely often*. Uma forma de encarar esse tipo de comportamento é pensar que para cada estado que φ é verdadeiro, deve haver um estado futuro para o qual φ deve ser verdadeiro novamente. Assim, φ ocorre *infinitely often* Fisher (2011).

Os seguintes exemplos, adaptados de Fisher (2011), representam possíveis utilizações desses conceitos para especificação de um sistema de alocação de impressoras compartilhada em rede. Para realizar uma impressão, determinado processo deve enviar uma mensagem, `imp_requerida(x)` para um servidor central. Este, por sua, envia um pedido de impressão `imprima(x)` para um dispositivo disponível. Cada mensagem e pedido de impressão deve ser acompanhado do nome do processo requerente, representado por `x`, para fins de identificação. Assim, possíveis propriedades desejadas para o sistema poderiam ser traduzidas em LTL, da seguinte forma:

Safety: $\mathtt{G}\neg(\mathtt{imprima(a)} \wedge \mathtt{imprima(b)})$ (1)
Liveness: $\mathtt{F(imprima(x))}$ (2)
Fairness: $\mathtt{GF(imp\_requerida(r))} \Rightarrow \mathtt{GF(imprima(r))}$ (3)

Propriedade (1) asserta que os processo `a` e `b` não devem imprimir ao mesmo tempo, ou seja um mecanismo de *Safety*. A segunda propriedade (2) diz respeito a *Liveness*, e asserta que eventualmente algo será impresso nesse sistema. Por fim, (3) trás o conceito de *Fairness*, assim, se um processo envia mensagens frequentes infinitas vezes é esperado que sejam realizados frequentes e infinitos pedidos de impressão.

2.3.2.2 *Model Checking*

Segundo Clarke (2008), o problema que impulsionou a criação de um método algorítmico para checar de forma exaustiva o espaço de estado do sistema, que por sua vez deu origem aos primeiros *Model Checkers*, pode ser estabelecido da seguinte forma:

> *"Se M for uma estrutura Kripke (i.e., um grafo de transição de estados), f uma expressão temporal (i.e., a especificação do sistema). Encontre todos os estados s de M de tal forma que, $M, s \models f$."*

Onde M segue a definição apresentada por Clarke e Emerson (1981):

$$M = (S, R, \pi)$$

onde, S é o conjunto de estados (S_0 sendo o estado inicial), R representa as relações de transição entre um estado e outro e π mapeia cada estado a um conjunto de proposições que são verdadeiras no respectivo estado.

A propriedade f do sistema, é geralmente expressa em uma lógica temporal. Inicialmente, CTL foi utilizado, mas como apontado posteriormente, não há vantagens em termos de complexidade em se utilizar CTL ou LTL (CLARKE, 2008). Em termos práticos, LTL possui menos operadores temporais, porém, características importantes de sistemas de *software* podem ser expressas de forma completa nessa lógica temporal. Ferramentas de sucesso como SPIN utilizam esse tipo de lógica para especificação (SPIN, 2015).

Outra forma de representar o funcionamento geral de um *Model Checker*, em termos de entradas e saídas, é mostrado na Figura 5. Uma importante característica de *Model Checking* é o poder de depuração através de contraexemplos. Quando encontrado um caminho de programa que não atende a especificação é retornado a sequência de estados que fizeram com que determinada propriedade fosse falsificada. Os contraexemplos fornecem uma forma de analisar possíveis falhas de implementação, compreender melhor suas origens, ou ainda caso seja detectado um problema, corrigir a especificação ou o modelo (implica em alterar a implementação) (ROCHA, 2015).

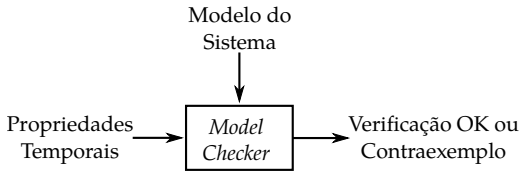

Figura 5 – Visão estrutural simplificada de *Model Checking*. Adaptado de (FISHER, 2011).

Em resumo, *Model Checking* foi introduzido para automatizar o processo de prova matemática do correto funcionamento de um sistema com relação a especificação. Isso é alcançado realizando a enumeração de todos os estados do sistema e aplicando um procedimento para decisão da validade das expressões temporais. Como vantagens desse método, deve-se citar: i) ao contrário de provas de teoremas, *Model Checking* é um método automático; ii) os contraexemplos são apropriados para depuração de sistemas complexos; iii) não é necessário uma especificação completa do sistema, o que faz com que *Model Checking* possa ser utilizado em fases iniciais de projeto.

Com relação às desvantagens é importante destacar o problema de explosão de estados, possivelmente o maior obstáculo para utilização em sistemas com complexidade considerável. Se for considerado um contador binário de n bits, por exemplo, o conjunto total de estados é dado por 2^n. Esse crescimento exponencial também é enfrentado em *Model Checking*, com relação ao número de processos e variáveis e, infelizmente, todas as implementações sofrem desse problema (CLARKE et al., 2012). O problema da explosão de estados tem motivado grande parte das pesquisas em *Model Checking* nos últimos anos, e levou ao desenvolvimento de abordagens como *Bounded Model Checking* (BMC), abordado na próxima seção.

2.3.2.3 *Bounded Model Checking*

Em *software* é comum a existência de caminhos de programa infinitos, ou que podem ser considerados infinitos dentro de um contexto. Nesses casos, *Model Checking*, na sua formulação inicial, sofre de problemas de explosão de estados e não é capaz de determinar a corretude de M com relação a f. Com os avanços obtidos nos solucionadores de *Satisfatibilidade* Booleana (SAT), a ideia de BMC foi introduzida para lidar com tais situações (BIERE, 2009). A ideia principal de BMC é limitar um caminho que talvez possa possuir um erro típico, a um prefixo finito conhecido. Com base nisso, o paradigma de BMC é baseado principalmente em falsificação de propriedades, o que permite escalonar muito melhor em comparação com *Model Checking* (BIERE, 2009). Segundo Cordeiro (2011), a ideia de BMC pode ser simplificadamente estabelecida da seguinte forma:

> *Dado um estrutura Kripke M e uma propriedade f, BMC desenlaça M, k vezes, e transforma em uma condição de verificação φ (uma propriedade safety) de forma que φ é satisfatível se e somente se f possui um contraexemplo de profundidade igual ou menor que k.*

Ao invés de realizar a busca por um contraexemplo considerando um valor estático para k, a grande maioria dos BMC utiliza de processo iterativo, em que k é aumentado gradativamente (BIERE, 2009). Algumas situações interessantes decorrem dessa ideia. Se φ for satisfatível: em uma situação idealizada (recursos ilimitados) um contraexemplo será encontrado; na situação de se esgotar os recursos nenhuma conclusão pode ser inferida. A situação mais interessante ocorre, no entanto, quando φ não é satisfatível. Nesse caso, algum critério deve ser utilizado para saber o momento de parar de aumentar k. Um dos métodos utilizados, baseado em teoria de grafos, é determinar o diâmetro do sistema, ou a maior distância entre estados conectados (BIERE, 2009). Utilizando teoria de grafos, é possível determinar um limite superior

para k, e assim, introduzir uma noção de completude, apesar de BMC não ser um método completo.

A principal vantagem de se utilizar BMC, em relação a um método *Model Checking* convencional, é justamente a capacidade em encontrar contraexemplos com maior facilidade de forma mais eficiente (BIERE, 2009). BMC é muito utilizado para esse propósito e se enquadra muito bem em metodologias de engenharia de *software*. Pelo fato de ser um método automático, assim como *Model Checking*, é preferível às metologias de testes manuais, embora muitas vezes possam ser integrados para aumentar cobertura de testes do sistema.

2.3.2.4 *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC)

Nessa seção, é descrito o funcionamento estrutural da ferramenta principal para verificação através de BMC utilizada nesse trabalho. As descrições apresentadas são fortemente baseadas nos trabalhos desenvolvidos por: Cordeiro (2011), Rocha (2015) e Morse (2015).

O ESBMC (*Efficient SMT-Based Context-Bounded Model Checker*) é um *Model Checker* para verificação de *software* ANSI-C[15] baseado em *Satisfiability Modulo Theories* (SMT).

Os passos principais realizados pelo ESBMC na verificação de um programa P são mostrados de forma simplificada na Figura 6. Inicialmente, o código é pré-processado para extração de um gráfico de controle de fluxo (GCF). Posteriormente, é gerado uma representação em formato `goto`, com desvios entre blocos de códigos gerados com base no GCF, o qual é obtido considerando o limite para desenlace k. Essa representação é convertida novamente para um programa constituído de *Single Static Assignments* (SSA) (cada variável é atribuída somente uma vez). No final desse processo o programa em SSA contém representações de todas as variáveis em todos os momentos de execução. Finalmente, o ESBMC converte o programa em SSA para uma *Quantifier-Free Formula* (QFF), expressões Booleanas sem quantificadores (e.g., $\exists$, $\forall$), entregue para um SMT *solver*, o qual decide a satisfatibilidade levando em conta uma combinação de teorias de base.

Do ponto de vista das condições de verificação, o ESBMC é capaz de gerar propriedades do tipo *safety* para: i) *underflow* e *overflow* em operações aritméticas, ii) limites de *arrays*, iii) segurança de ponteiros e iv) alocação de memória dinâmica.

[15] Padrão estabelecido pelo *American National Standards Institute* (ANSI) para linguagem de programação C.

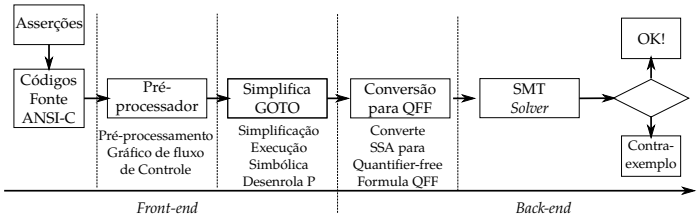

Figura 6 – Visão estrutural do ESBMC. Adaptador de (CORDEIRO, 2011).

2.4 Conclusões e Discussão

A complexidade dos sistemas embarcados vem aumentando gradativamente nas últimas décadas o que reflete em uma dificuldade crescente no desenvolvimento. Assim, os métodos utilizados para implementação se adequam constantemente para fornecer melhor suporte. Duas metodologias de desenvolvimento de sistemas embarcados se destacam recentemente: *Model Based Design* e *Electronic System Level Design*. MBD é mais utilizada para modelagem de elementos dinâmicos, algoritmos de controle ou processamento de sinais. Por outro lado, ESL tem seu ponto forte em modelagem de *software* e *hardware*, com uma tendência recente em utilização de plataformas virtuais para aceleração do processo de desenvolvimento. Essas metodologias podem ser utilizadas independentemente, porém a combinação das duas tem o potencial de aumentar ainda mais a produtividade, considerando também o processo de verificação de *software*.

Recentemente, muitos avanços em *Model Checking* tornam necessário considerar essa técnica para verificação de *software* ainda em fases iniciais de projeto, tendo em mente que não é necessário uma especificação completa para sua utilização. Embora os recentes avanços, o problema de explosão de estados ainda é uma realidade considerando sistemas complexos. Nesses casos, é necessário alguma forma de testar o sistema através de algum mecanismo. A perspectiva mais importante para este trabalho é a exploração e combinação de diversas técnicas de desenvolvimento e verificação de *software* para redução do tempo total de projeto e alcançar maior cobertura de testes do sistema.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta o estado da arte relacionado com as principais áreas de pesquisa dentro do contexto desse trabalho.

Os trabalhos que serão discutidos foram coletados através de técnicas de pesquisa *snowballing*[1]. Com base nisso, é de conhecimento do autor que os trabalhos apresentados nesse capítulo constituem uma pequena parcela do estado da arte, porém já são suficientes para suportar a argumentação desse trabalho.

No restante desse capítulo, é apresentado o estado da arte relacionado a: Verificação Formal com *Model Checking* (análises estáticas), Verificação por Simulação (i.e. Teste ou análises dinâmicas) e Métodos Híbridos (combinação de *Model Checking* e simulação).

3.1 Verificação Formal: *Model Checking*

Desde o desenvolvimento do primeiro[2] *model checker* em 1983, muitos trabalhos e ferramentas foram desenvolvidos nessa área. Atualmente, diversos *model checkers* compõem o estado da arte em verificação formal de *software*, cada um com determinada especialidade.

3.1.1 Atual Panorama de *Model Checking*

A SV-COMP (Competição em Verificação de *Software*) é uma competição realizada anualmente que tem por objetivo sintetizar o estado da arte em *Model Checking* e fornecer um conjunto de *benchmarks* para possíveis comparações entre diferentes ferramentas. O formato da competição é centrado em um conjunto de tarefas de verificação[3], classificadas segundo funcionalidade. Os participantes podem escolher em qual categoria participar, de forma que para comparação e atribuição de ganhadores, sendo avaliados e premiados pela categoria e geral.

Na Tabela 2, são sintetizados os resultados finais da edição 2015 da competição. Partindo do princípio de que o sistema de pontuação utilizado é

[1] Termo utilizado para denominar o tipo de pesquisa, que retorna a partir de uma população inicial (artigo ou trabalho importante em determinada área), uma nova população relacionada. Geralmente a partir do conjunto de referências da população inicial.

[2] O algoritmo original foi concebido simultaneamente por Clarke e Emerson (1981) e Queille e Sifakis (1982) ainda na década de 80. O primeiro *model checker* foi implementado somente em 1983 por Clarke, Emerson e Sistla (1983).

[3] As tarefas são executadas em um servidor Linux com 4 núcleos de processamento até 3,4 GHz. Dois limites são estabelecidos: tempo, 15 minutos e memória 14,6 GB.

justo (ferramentas diferentes são analisadas considerando critérios coerentes para comparação), pode-se tirar algumas conclusões com relação a esses resultados. Comparando a pontuação máxima de cada categoria com a pontuação atribuída, dois extremos são evidentes; o pior resultado foi na categoria *Array* (programas envolvendo o processamento de arranjos) com apenas 33,1% do total de pontos máximos; o melhor resultado, por outro lado, foi na categoria *Concurrency* (programas que envolvem programação concorrente) em que 100% dos pontos foram alcançados. Esses resultados fornecem evidência que as ferramentas ganhadoras possuem maior eficácia na verificação dessas categorias, em comparação com as outras ferramentas. A análise mais interessante, para fins de argumentação nesse trabalho, é com relação ao ganhador geral (CpaChecker). Do total de pontos (9519), somente 51,36% (4889) foi alcançado pelo ganhador, ou seja, de todas as tarefas (5803) somente 3211 foram verificadas corretamente pela ferramenta que obteve o melhor desempenho geral. Esse valor corresponde a 55,33% das tarefas totais.[4] A diferença entre a pontuação total e a percentagem de tarefas verificadas corretamente diz respeito ao sistema de pontuação adotado, que penaliza respostas incorretas.

Mesmo considerando que muitas dessas tarefas de verificação funcionam como desafios para as ferramentas, pois estimulam características específicas de forma intensiva, o número total de tarefas verificadas corretamente está muito além de ter uma cobertura significativa às mais diversas classes de programa. Muitos avanços foram realizados na área de verificação através de *Model Checking* ao longo dos últimos anos de pesquisa, embora as limitações, como explosão de estados, ainda restrinjam a utilização em sistemas de complexidade considerável, mesmo para as melhores ferramentas em *Model Checking* do estado da arte.

3.1.2 Verificação de Propriedades Temporais Usando o ESBMC

Uma contribuição recente para o estado da arte em verificação de propriedades temporais com BMC foi apresentada por Morse et al. (2013). Nesse trabalho, são introduzidas alterações à ferramenta ESBMC que permitem a verificação de propriedades mesmo considerando *bounded traces*. Os autores afirmam que esse trabalho é o primeiro a implementar tal mecanismo considerando código C sem modificações.

Para alcançar tal objetivo, o ESBMC desenlaça possíveis laços infinitos e intercala a execução simbólica do *bounded trace* com um monitor em C, criado através de uma modificação da ferramenta `ltl2ba`[5], chamada de `ltl2c` pelos

[4] Em resultados mais recentes (2016), o valor atingido na categoria *Array* foi cerca de 60.1%, concorrência 100%, e geral (ganhador UAutomizer) cerca de 44.6%.

[5] <http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/>

Tabela 2 – Ganhadores em cada categoria da competição SV-COMP 2015. Adaptado de (SV-COMP, 2015).

Model Checker	Array	Bit Vectors	Concurrency	ControlFlow	Device Driver	Floats	Heap Manipulation	Memory Safe	Recursive	Sequentialized	Simple	Termination	**Overall**
Tarefas de verificação	86	47	1003	1927	1650	81	80	205	24	261	46	393	5803
Pontuação Máxima	145	83	1222	3122	3097	140	135	361	40	364	68	742	9519
AProVE	–	–	–	–	–	–	–	–	–	–	–	610	–
Beagle	–	4	–	–	–	–	–	–	6	–	–	–	–
BLAST 2.7.3	–	–	–	983	2736	–	–	–	–	–	32	–	–
Cascade	–	52	–	537	–	–	70	200	–	–	–	–	–
CBMC	-134	68	1039	158	2293	129	100	-433	0	-171	51	–	1731
CPAchecker	2	58	0	2317	2572	78	96	326	16	130	54	0	4889
CPArec	–	–	–	–	–	–	–	–	18	–	–	–	–
ESBMC 1.24.1	-206	69	1014	1968	2281	-12	79	–	–	193	29	–	-2161
FOREST	–	–	–	–	–	–	–	–	–	–	–	–	–
Forester	–	–	–	–	–	–	32	22	–	–	–	–	–
FuncTion	–	–	–	–	–	–	–	–	–	–	–	350	–
HIPTNT+	–	–	–	–	–	–	–	–	–	–	–	545	–
Lazy-CSeq	–	–	1222	–	–	–	–	–	–	–	–	–	–
Map2Check	–	–	–	–	–	–	–	28	–	–	–	–	–
MU-CSeq	–	–	1222	–	–	–	–	–	–	–	–	–	–
Perentie	–	–	–	–	–	–	–	–	–	–	–	–	–
Predator	–	–	–	–	–	–	111	221	–	–	–	–	–
SeaHorn	0	-80	-8973	2169	2657	-164	-37	0	-88	-59	65	0	-6228
SMACK+Corral	48	–	–	1691	2507	–	109	–	27	–	51	–	–
Ultimate Automizer	2	5	–	1887	274	–	84	95	25	15	0	565	2301
Ultimate Kojak	2	-62	–	872	82	–	84	66	10	-10	3	–	231
Unbounded Lazy-CSeq	–	–	984	–	–	–	–	–	–	–	–	–	–
Ganhador	**48**	69	**1222**	2317	2736	129	111	326	27	193	65	610	**4889**
Pontos Obtidos / Pontos Totais	33.10%	83.13%	100.00%	74.22%	88.34%	92.14%	82.22%	90.30%	67.50%	53.02%	95.59%	82.21%	51.36%

autores. Do ponto de vista de utilização são definidas propriedades em LTL, as quais são otimizadas e convertidas em autômatos em C, do tipo Büchi, pela ferramenta `ltl2c`.

A validade de determinada propriedade, considerando *traces* de programa finitos, acaba sendo influenciada pelo limite de desenlace considerado. Como exemplo demonstrativo, Morse et al. (2013) utiliza os códigos mostrados no Programa 1.

Programa 1 Trechos de programa com *traces* infinitos iguais. Adaptado de (MORSE et al., 2013).

P1

```
1  int s = 0;
2  while(true){
3      s = 1 - s;
4  }
```

P2

```
1  int s = 0;
2  while(true){
3      s = 1;
4      s = 0;
5  }
```

P3

```
1  int s = 0;
2  s = 1;
3  while (true) {
4      s = 0;
5      s = 1;
6  }
```

Os três programas apresentam o mesmo comportamento, alternando `s` de 0 para 1 infinitamente, seguindo a ideia de *infinitely often*. Na situação de se avaliar $G(\{s == 0\} \Rightarrow F\{s == 1\})$, em *traces* infinitos, a propriedade sempre se mantém verdadeira para os três programas. No entanto, para *traces* finitos, o resultado é dependente do limite de desenlace e a validade da propriedade pode mudar para os três programas. Considerando `P1`, ver Programa 1; se o limite de desenlace é determinado para um valor ímpar, implicaria em `s==1` (há resposta para o pedido `s==0`); porém se for o limite for par, `P1` terminaria com `s==0` (não há resposta para o pedido). `P2` sempre termina com `s==0` independente do limite de desenlace, enquanto `P3` sempre com `s==1`.

Esses cenários podem levar a diferentes resultados com base no limite de desenlace do programa. Como consequência, a ferramenta retorna 3 possíveis resultados da avaliação de propriedades: verdadeiro, falso e possivelmente falso. Uma das desvantagens para utilização dessa abordagem diz respeito a essa dependência. Por outro lado, a sua aplicação pode ser feita de forma modular, considerando recortes de código, o que afirma sua aplicabilidade em fases iniciais de projeto.

3.2 Verificação por Simulação

Teste é um dos tópicos mais pesquisados e talvez um dos mais antigos em engenharia de *software*. Para garantir qualidade do produto final, testes são conduzidos ao longo de todo o ciclo de desenvolvimento e podem corresponder à grande parte do tempo total do projeto (ORSO; ROTHERMEL, 2014). Essas

razões dificultam a discussão com relação ao estado da arte em teste de *software*, devido principalmente ao grande número de pesquisas e publicações realizadas recentemente.

Assim, é importante definir um escopo e identificar as recentes tendências na área. Um trabalho que destaca os principais avanços em teste de *software* foi desenvolvido por Orso e Rothermel (2014). Nesse trabalho, foi conduzido uma pesquisa com 50 pesquisadores na área, em uma tentativa de identificar quais foram os maiores avanços e quais são os desafios futuros em teste de *software*, com base no período 2000-2014. Os resultados revelam vários campos de pesquisa e uma forma resumida de estado da arte em alguns dos assuntos mais recorrentes. É importante destacar que o foco dos autores é majoritariamente na investigação de *software* de propósitos gerais, que não corresponde ao foco deste trabalho. Porém, é evidente que muitas dessas ideias acabam sendo rearranjadas futuramente para outras aplicações e áreas, e.g., aplicações embarcadas. Esse motivo reforça a ideia de realizar uma discussão geral focada em conceitos abrangentes e não tão focado em métodos específicos.

Duas das áreas que avançaram mais nos últimos anos segundo Orso e Rothermel (2014) foram: Geração Automática de Testes e Estratégias de Teste. Além disso, alguns dos maiores desafios estariam na utilização dos métodos em sistemas atuais reais e domínios específicos de aplicação, tais como abordados nesse trabalho.

Nas próximas seções, são resumidas algumas das informações mais relevantes apresentadas por Orso e Rothermel (2014). Como essas informações são fortemente baseadas nesse trabalho, citações repetidas dos mesmos autores serão evitadas.

3.2.1 Geração Automática de Testes

Geração automática de testes é um dos assuntos mais recorrentes em publicações sobre teste de *software*, especialmente considerando execução simbólica e testes randômicos. Por mais que esse assunto não seja explorado diretamente nesse trabalho, é importante destacar os recentes avanços nessa área, pois afinal faz parte do estado da arte em teste de *software*.

3.2.1.1 Execução Simbólica

Na realização de testes através de execução simbólica, o programa é constituído por estados simbólicos expressos em função das entradas, as quais, são representadas como uma série de restrições em forma normal conjuntiva. Em termos práticos, pode-se utilizar execução simbólica para determinar uma entrada (conjunto de variáveis) que faz com que certo caminho ou ponto do

código seja alcançado. Esse tipo de teste é importante especialmente para atingir altos níveis de cobertura de código.

As restrições de entradas são entregues a um solucionador que determina os valores para os quais o objetivo inicial é cumprido. Um problema comum, que define um dos tipos mais utilizados de execução simbólica, ocorre quando o solucionador não é capaz de resolver as restrições dadas para as entradas.

Uma forma de lidar com esse problema e um dos tópicos mais explorados dentro do contexto de execução simbólica é *Dynamic Symbolic Execution* (DSE). DSE utiliza uma combinação de execução concreta e simbólica (*concolic execution*) para extrapolar as limitações do solucionador utilizado. Assim, execução simbólica e concreta são conduzidas ao mesmo tempo durante o processo de teste. Desta forma, quando o solucionador falha em fornecer uma resposta, os valores são combinados e a restrição pode ser simplificada. Embora o recente interesse em pesquisa nessa área, ainda há grandes limitações com relação a complexidade do sistema, em particular sistemas com entradas de dados altamente estruturadas.

Diversas ferramentas destacam-se nessa área, tais como Klee[6] e Crest[7], que muitas vezes utilizam o termo testes concólicos para se referir a geração automática e execução simbólica.

A aplicação dessas ferramentas e execução simbólica a *software* embarcado pode ser relativamente restrita, especialmente considerando construtores de baixo nível. É evidente que dependendo da estratégia utilizada para se verificar o sistema diferentes técnicas são empregadas. Execução simbólica apresenta maiores vantagens em aumento de cobertura de código e também, em testes específicos com intuito de estimular porções específicas do código.

3.2.1.2 Testes Randômicos

Como definido na Seção 2.3.1 o processo de teste é essencialmente baseado na definição dos casos de teste T, levando em conta o *domínio* de programa $D(P)$. Quando a definição de T ocorre de forma randômica, diz-se que o teste é do tipo randômico. Por mais simples que a ideia se apresente, esforços consideráveis vem sendo aplicados nessa área, em especial para aumentar sua eficiência geral ou lidar com $D(P)$ de forma eficiente.

Um dos campos de pesquisa mais promissores em testes randômicos é aplicação de algoritmos adaptativos para seleção de novos candidatos de teste (comumente denominado *Adaptive Random Testing* (ART)). A fundamentação

[6] <https://klee.github.io/>
[7] <http://jburnim.github.io/crest/>

dessa ideia é heurística, de forma que a partir de um conjunto de valores iniciais, selecionados randomicamente, são selecionados novos conjuntos "distantes", de acordo com alguma heurística de seleção. Alguns trabalhos mostram que ART pode ser mais rápido em expor *bugs* que métodos randômicos convencionais (e.g. (TAPPENDEN; MILLER, 2009) e (LIN et al., 2009)).

Claramente, a desvantagem da utilização de testes randômicos é cobertura; considerando a complexidade do sistema ($D(P)$ pode ser gigantesco), torna assim, impraticável teste de todos os valores possíveis. Com base nisso, alguns trabalhos propõem a combinação de randomização dos valores de testes com execução simbólica (GODEFROID; KLARLUND; SEN, 2005).

3.2.2 Estratégias de Testes

Nessa seção, são apresentados alguns resultados recentes e um panorama geral de certas estratégias para teste. A denominação *estratégia* é utilizada para separar de métodos específicos para execução e determinação de entradas, introduzindo assim, uma concepção mais geral sobre outras alternativas que constituem o estado da arte em testes de *software*.

3.2.2.1 Testes Combinatórios

Em se tratando de *software* de propósito geral é comum uma grande variedade de configurações e parametrizações diferentes para atender a diferentes cenários. Uma estratégia adotada para garantir qualidade envolve testes combinatórios considerando as implicações dessas configurações no comportamento do *software*.

Evidentemente que a complexidade do sistema tem papel crucial na explosão combinatorial, o que torna a execução de testes em todas as configurações possíveis impraticável em sistemas modernos. Apesar dessa limitação, esse assunto vem sendo largamente explorado.

Segundo levantamento realizado por Nie e Leung (2011) de 97 artigos publicados entre 1985 e 2008, 77 apareceram após os anos 2000. Considerando somente o ano de 2015 e a base de dados do *IEEE Xplore*, uma rápida pesquisa para a *string "Combinatorial Testing"* retorna 26 trabalhos com essa *string* no título; dois dos quais foram publicados em *Journals* e 24 em conferências. Dentre essas publicações se destaca os resultados de um estudo de dois anos conduzido por uma iniciativa da empresa da área aeroespacial *Lockheed Martin* (HAGAR et al., 2015). Nesse estudo, testes combinatoriais foram introduzidos em diversos projetos pilotos, que acabaram por reportar redução significativa em custos de desenvolvimento e uma melhora de cobertura de teste de 20% a 50%.

A perspectiva de *software* embarcado pode ser um pouco diferenciada, visto que grande parte dos sistemas são *ad hoc* e não possuem muitas parametrizações ou configurações diferenciadas. No entanto, testes combinatoriais podem ser empregados para explorar diferentes interações entre módulos e cenários específicos, em que há parâmetros de entrada tais como *hardware*, sinais e sensores (HAGAR et al., 2015).

3.2.2.2 Testes Baseados em Modelos

A tendência geral de elevar o nível de abstração para lidar com sistemas cada vez mais complexos introduz novas ferramentas de modelagem e desenvolvimento à engenharia de *software*. É o caso de diagramas de usos de caso, troca de mensagens, ou notações baseadas em estados, como modelos de estados finitos, gráficos de estados e diagramas de sequência. Além disso, ainda há ferramentas e padrões voltados para modelagem estrutural como *Unified Modeling Language* (UML).

Seguindo essa mesma tendência certos tipos de testes de *software* migraram para níveis mais abstratos, considerando não o sistema final, mas um modelo. Essa tendência foi agrupada por Orso e Rothermel (2014) como *Model-Based Testing*, ou testes baseados em modelo.

Devido em grande parte ao sucesso da adoção das ferramentas de modelagem, testes baseados em modelos são muito utilizados na indústria. Associado a isso, certas facilidades encontradas em níveis mais abstratos de teste auxiliam sua aceitação.

Como desvantagem desse método, deve ser citado a necessidade em se utilizar um fluxo de desenvolvimento baseado em modelos, ou criar o modelo em algum momento, o que pode ser uma atividade relativamente intensiva e demorada. Novamente, a complexidade do sistema pode ser um fato significativo e limitante para aplicação de teste baseados em modelos, mesmo em níveis mais abstratos.

3.2.3 Desafios Futuros em Teste de *Software*

Dentre os desafios apresentados por Orso e Rothermel (2014), é importante destacar testes em domínios específicos, que nada mais é que aplicações de testes a casos e paradigmas particulares.

No contexto desse trabalho, o domínio específico é justamente *software* embarcado baseado na conjunção de abstrações ou interfaces entre *software* e *hardware*. Para fins de esclarecimento, considere um programa escrito na linguagem C, como apresentado no Programa 2. O funcionamento é bem simplificado, a cada determinado intervalo de tempo (predefinido por algum

mecanismo de escrita em um registrador de controle, normalmente mapeado em memória) esse programa pisca um *led* toda vez que a função *Interrupt Service Routine* `ISR()` é chamada.

Do ponto de vista da semântica da linguagem C esse programa não é válido, pois a função `ISR()` não é chamada nenhuma vez no corpo da função `main()`. Na realidade, um construtor específico:

```
#pragma interrupt_handler ISR
```

deve ser usado para evitar que o compilador otimize e exclua essa função. No entanto, quando analisamos o conjunto *hardware* e *software*, pensando em uma arquitetura específica, esse programa pode funcionar, se a implementação estiver correta. Em consequência, não há ferramenta capaz de analisar esse código ou testá-lo sem levar em conta o funcionamento e implementação do *hardware* em questão.

Programa 2 Exemplo simplificado de programa com interrupção por tempo.

```
1  #pragma interrupt_handler ISR
2  void ISR() {
3    blink_led();
4  }
5
6  void main() {
7    // configuração registradores timer
8    while (1);
9  }
```

Embora, grande parte do *software* escrito para propósitos gerais não possua esse tipo de implementação, isso é muito comum em *software* embarcado e define, como exemplo, um domínio de aplicação que apresenta limitações adicionais para as atuais ferramentas de análise e teste do estado da arte.

Os benefícios da utilização de plataformas virtuais para o desenvolvimento pode ser associado a esses domínios de aplicação. Fornecendo, assim, uma camada de isolamento para simulação e possíveis testes e validação desse tipo de aplicação. Apesar de existir algumas ferramentas comerciais que fornecem suporte parcial a esse tipo desenvolvimento e teste, há muitos trabalhos e pesquisas a serem realizadas nessa área, especialmente demonstrações de integração entre ferramentas de análises com desenvolvimento e plataformas virtuais, ou seja, a combinação em abordagens híbridas.

3.3 Métodos Híbridos

Nas seções anteriores desse capítulo, foram relacionados alguns trabalhos recentes que fazem parte de duas áreas, dentro do contexto de verificação de *software*. Como comentado previamente, ambas abordagens (*Model Checking* e Simulação/Teste), possuem sérias limitações quando aplicadas a sistemas complexos modernos; as quais tem incentivado grande parte das pesquisas recentes na busca por melhorias nos métodos e ferramentas. Por outro lado, alguns poucos trabalhos exploram a integração de *Model Checking* e teste em fluxos de desenvolvimento em alto nível (e.g., baseados em plataformas virtuais), voltados para verificação de *software*. Apesar de haver certo suporte por parte de ferramentas comerciais voltadas para verificação de *hardware*, ainda há uma lacuna de trabalhos nesse sentido.

Um dos primeiros trabalhos que demonstra a integração de simuladores, ou modelos em alguma linguagem de descrição de *hardware*, com um fluxo de simulação/teste de *software* ou co-verificação (*hardware* e *software*) foi apresentado por Lettnin et al. (2007). Esse trabalho foi desenvolvido usando as ferramentas comerciais da empresa *Cadence Design Systems*. Na realidade é uma adaptação dos mecanismos de verificação de *hardware*. A aplicação demonstrada utiliza um modelo em SystemC de um processador PowerPC 750, o qual é adaptado para fornecer acesso e sincronização as variáveis e funções do *software* embarcado. Essa abordagem ainda está disponível nessas ferramentas, porém apresenta sérias limitações. A mais crítica das limitações é manter sincronia entre os dois domínios diferentes. Por um lado, um ambiente totalmente desenvolvido para verificação de *hardware* (*Specman* e a linguagem ***e*** (Cadence Design Systems, 2016)), e pelo outro, modelos que podem ser implementados em SystemC de forma puramente algorítmica, sem informação nenhuma de tempo. Além disso, modificações no *software* embarcado são necessárias, justamente para facilitar o processo de sincronização. Mesmo assim, essa metodologia foi uma das primeiras a apresentar suporte para integração de um simulador de um conjunto de instruções com diversos periféricos, constituindo uma plataforma virtual, em um fluxo de verificação. Uma das principais vantagens, do ponto de vista de verificação, seria a utilização dos instrumentos fornecidos pela Linguagem e, para cobertura de valores e de testes.

Do ponto de vista de integração entre análises dinâmicas e estáticas, dois trabalhos se destacam.

Cordeiro et al. (2009) propôs uma abordagem híbrida para verificação de *software* embarcado, considerando a interface entre *hardware* e *software*. Segundo os autores, essa abordagem pode fornecer maior cobertura e redução significativa do tempo de verificação. O fluxo inicia com a definição de uni-

dades de testes, que devem ser implementadas e compiladas antes do início do desenvolvimento da aplicação final. Juntamente com a criação dessas unidades, um *backlog* do produto é criado e realimentado de forma incremental, para manter uma lista de requerimentos do produto. O procedimento prático de verificação seria baseado na integração de descrições do processador e seus periféricos (ambos criados na linguagem *Verilog*), com o *software* sendo verificado. O conjunto, *software* e descrições *Verilog*, seria traduzido em uma representação em *Binary Decision Diagrams* (BDDs) ou SAT, para posterior exploração de propriedades em LTL, CTL ou *Property Specification Language* (PSL). Esse modelo final, dependendo da complexidade do sistema, é composto por diversos elementos que podem mascarar um erro do *software*. Dessa forma, Cordeiro et al. (2009) propõe três abordagens diferentes para evitar tais situações: (1ª) verificar somente os elementos que não utilizam de interface com *hardware*, (2ª) verificar novamente considerando a interface *hardware*/*software* e a (3ª) realizar testes automáticos, com base nos contraexemplos gerados por *Model Checking*, utilizando o modelo em *Verilog*. As vantagens e desvantagens dessa abordagem estão relacionadas a utilização do modelo em *Verilog*. Por um lado há diversos detalhes com relação a temporização, e análises quantitativas podem ser realizadas. Em contrapartida, dependendo das propriedades de interesse em verificar, informações desnecessárias podem tornar o sistema muito lento para simulação, comparado a modelos mais abstratos, e.g., modelos em SystemC. Assim, simulação somente é usada para realizar testes caso um dos *model checkers* utilizados retorne um contraexemplo, não há uma colaboração explicita entre as duas abordagens.

Um trabalho também publicado durante o mesmo período mostra outro ponto de vista de integração híbrida. Lettnin (2009) elaborou uma heurística para integração entre *Model Checking* e simulação de forma a complementar a cobertura de estados do sistema e, assim, lidar com problemas que podem decorrer da explosão de estados. Nessa metodologia, são extraídos um modelo formal e um modelo para simulação do programa sob verificação. O modelo formal é usado para análises estáticas, enquanto o modelo para simulação pode ser utilizado em uma implementação em C (executando diretamente em modelo de um processador (e.g., em SystemC)) ou em SystemC. A combinação entre as análises é baseada no conhecimento desses modelos e no formato definido de troca de informações e propriedades, entre simulação e o modelo formal. A ideia principal é fazer uma mudança de contexto entre simulação e formal (e vice e versa), no momento em que determinada função ou estado é alcançado, ou quando um limite de memória foi atingido pelo *model checker*. Os pontos fortes dessa metodologia residem na utilização da heurística para combinação entre *model checking* e simulação, também na automação de grande

parte dos passos no processo de verificação.

Em suma, as limitações apresentadas pelos *model checkers* atuais podem ser superadas com a utilização de metodologias híbridas. Essas abordagens não trazem garantias quanto a presença ou ausência de falhas na implementação, somente fornecem mecanismos para estressar o sistema em diversos pontos de operação, e assim, cobrir um maior espaço de estados. Analisando os trabalhos relacionados até o momento, e mais recentes (e.g. (BEHREND et al., 2015)), ainda há uma lacuna em abordagens que explorem o desenvolvimento de modelos e plataformas virtuais em cenários distintos de combinação entre *Model Checking* e simulação. É justamente nessa lacuna que este trabalho esta localizado, como uma exploração de diferentes cenários de teste, verificação formal e desenvolvimento com plataformas virtuais.

3.4 Conclusões e Discussão

Neste capítulo, foram discutidos alguns trabalhos do estado da arte que estão relacionados com os tópicos abordados nesse texto. Diversos dos quais, ainda estão sujeitos a avanços significativos, visto que, são áreas de pesquisa relativamente ativas na atualidade.

Dentro do contexto de métodos híbridos de verificação, há uma pequena quantidade de trabalhos publicados, apesar de ser uma prática comum na indústria e em verificação de *hardware*. Nessa mesma área, não há nenhum trabalho, de conhecimento do autor, que explore a combinação de metodologias de desenvolvimento (e.g. MBD e ESL) e plataformas virtuais para fins de verificação de *software*. Considerando essa perspectiva, fica claro a importância e posicionamento desse trabalho perante ao estado da arte.

4 METODOLOGIA PROPOSTA

A metodologia para avaliação dos diferentes cenários de combinação entre *Model Checking*, simulação e desenvolvimento é descrita nesse capítulo. A descrição será realizada inicialmente de forma mais abrangente, com detalhamentos posteriores para cada etapa da metodologia.

4.1 Descrição Geral da Metodologia

A Figura 7 representa de forma sintetizada as principais etapas da metodologia descrita neste capítulo. Em comparação com desenvolvimento convencional ou tradicional, essa metodologia fornece a possibilidade de iniciar o desenvolvimento e verificação do sistema ainda antes do *hardware* final estar pronto. Para atingir essa meta, uma etapa inicial de implementações é necessária, dando suporte ao projeto do *software* embarcado.

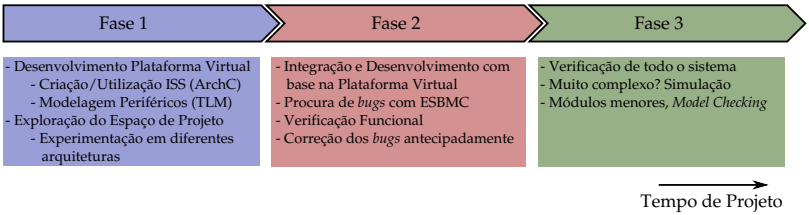

Figura 7 – Visão esquemática das três principais fases da metodologia utilizada com suas principais atividades.

A Fase 1 (ver Figura 7) é baseada na metodologia ESL, ou seja, deve-se criar uma série de modelos em SystemC, os quais, serão posteriormente incorporados em uma plataforma virtual. Nessa etapa, há liberdade total para explorar arquiteturas e configurações diferentes para o sistema, dependendo somente da disponibilidade e interesse em desenvolvimento dos modelos.

Considerando uma plataforma virtual constituída por um ou mais processadores (processamento homogêneo ou heterogêneo), e periféricos que são relevantes na interface *hardware*/*software*, duas perspectivas se definem na Fase 1: deve-se modelar os simuladores de conjunto de instruções, que são os elementos centrais na plataforma; outro ponto de vista diz respeito a modelagem dos periféricos necessários.

A criação dos simuladores, como mencionado anteriormente, pode ser uma tarefa parcialmente automatizada a partir de descrições abstratas. LDAs, como ArchC, cumprem papel essencial no auxílio e aceleração desse processo. Por parte dos periféricos, algumas alternativas estão disponíveis

atualmente; no contexto de SystemC, *Transaction Level Modeling* é melhor opção, pois abstrai informações desnecessárias e permite modelagem algorítmica. Algumas ferramentas fornecem possibilidade para criação e geração automática de modelos em SystemC/TLM a partir de descrições a nível de *hardware* (e.g. `HIFSuite`[1]); outras são especializas na criação de produtos e plataformas virtuais SystemC/C++ (e.g. `GreenSocs`[2]).

Diversas são as vantagens em desempenhar certo tempo de projeto nessa etapa inicial (Fase 1). Da perspectiva de desenvolvimento do *software* embarcado, por exemplo, o processo de depuração pode utilizar emulação de chamadas do sistema operacional (i.e., *System-Call Emulation*) para auxiliar no processo de depuração. Além disso, um ambiente virtual fornece maior maleabilidade para realizar testes e experimentações. Apesar dessas vantagens, essa etapa inicial de implementações pode ser árdua e tomar grande parte do tempo de projeto.

Do ponto de vista de verificação, deve-se garantir o correto funcionamento da plataforma virtual. Para isso, monitoramento de propriedades durante a execução pode ser uma possibilidade a ser considerada. Abordagens que automatizam parte do processo são escolhas pertinentes pois tem o potencial de reduzir significativamente a quantidade de erros inseridos durante programação.

A Fase 2 é destinada a implementação do *software* embarcado, que é realizada com base nas aplicações e modelos criados na etapa anterior. Muitos cenários diferentes podem surgir, pois são muitas combinações e possibilidades de integração entre os modelos e a aplicação final sendo desenvolvida.

É considerado também nessa etapa, a possibilidade de iniciar o processo de verificação com BMC. Mesmo que o código não esteja completo, pode-se verificar cada função separadamente, em busca por possíveis erros na implementação. *Model Checkers* podem funcionar também como ferramentas para depuração do código, principalmente com auxílio de simulações, e assim, auxiliar no processo como um todo (CLARKE, 2008).

Durante as simulações o desenvolvedor pode coletar informações relevantes sobre o sistema e com base nisso, auxiliar a redução do espaço de procura pelo *model checker*. Esse cenário é interessante, pois pode fornecer alternativas quando limites de memória ou tempo são atingidos pelo *model checker*. É possível também realizar investigações funcionais com relação ao comportamento do sistema. O ESBMC fornece a opção de verificação de propriedades LTL através de monitores. Essas mesmas propriedades podem ser futuramente monitoradas durantes execuções, usando mecanismos simples de

[1] <http://www.hifsuite.com/>
[2] <http://www.greensocs.com/>

adaptação dos monitores.

De forma geral, a combinação de desenvolvimento baseado em plataformas virtuais com *model checking*, permite construir o sistema de forma incremental e reduzir a quantidade de erros finais. Tanto o *model checker* quanto a plataforma virtual podem oferecer limitações durante essa etapa de projeto. É importante ter em mente que o simples fato de exercitar as especificações, e analisar o sistema com maior rigorosidade pode revelar erros ainda em etapas iniciais de projeto (BAIER; KATOEN, 2008).

Na Fase 3, o sistema é integrado em sua totalidade. Todos os módulos, processadores, periféricos e o *software* embarcado, sendo executados nos modelos, são considerados para testes e validação. Devido à complexidade, é relativamente difícil aplicar *model checking* para verificação do sistema completo de forma satisfatória. Nesses casos, pode-se considerar mecanismos de monitoramento de propriedades para realizar simulações específicas, e dessa forma contribuir para estressar o sistema em diferentes pontos operativos. No entanto, se cada módulo/função foi corretamente analisado na etapa anterior é esperado que grande parte dos erros introduzidos durante a programação, já tenham sido detectados e corrigidos anteriormente.

As vantagens da utilização dessa metodologia podem não ser tão evidentes, sobretudo quanto ao tempo de projeto. Se por um lado, desenvolvimento e verificação podem iniciar muito antes, certo tempo deve ser dedicado exclusivamente as implementações iniciais da plataforma virtual e modelos dos sistema. Por esse motivo, há empresas fornecendo soluções para auxiliar nessa área, justamente para dar suporte e agilizar esse processo inicial. Em suma, essa metodologia não trás conceitos ou definições novas, tudo que foi apresentado nessa seção já é de conhecimento. O enfoque deste trabalho é na demonstração e exploração dessa metodologia, dentro do contexto específico das ferramentas e implementações consideradas, e seu potencial de aceleração do processo de desenvolvimento e verificação.

4.2 Relação entre as Fases da Metodologia Utilizada

As principais atividades que compõe a metodologia foram estabelecidas e brevemente descritas na seção anterior. Nesta seção, será descrito a relação entre cada passo da metodologia com base no fluxograma mostrado na Figura 8.

A primeira etapa, a partir do início do projeto, é a definição das especificações e requisitos que o sistema sendo desenvolvido deve atender. Há inúmeras formas de se especificar um sistema, porém a única preocupação deste trabalho é com propriedades funcionais (i.e., expressas em lógica tem-

[3] A simbologia utilizada para os fluxogramas é mostrada Apêndice C.

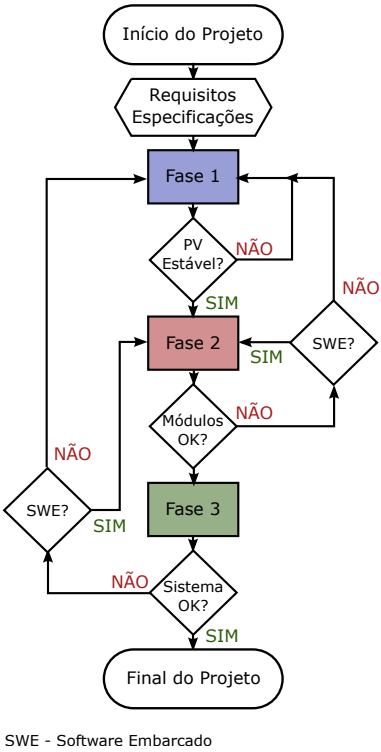

Figura 8 – Fluxograma representando a relação entre as fases da metodologia utilizada[3].

poral). Essa etapa de especificação é tão importante quanto o projeto em si e, normalmente, há realimentação de informações no intuito de manter atualidade e consistência. Como pode ser visualizado na Figura 8, esse processo de realimentação não é representado. A intenção é demonstrar um diagrama do ponto de vista de verificação. Assim, conceitualmente, não deve haver mudanças na especificação do sistema, uma vez que isso pode mascarar possíveis erros de implementação. Divergências entre a especificação e a implementação sendo verificada são desconsideradas para este trabalho, ou seja, a especificação sempre é tomada como correta.

Uma vez definida as especificações, pode-se iniciar as implementações na Fase 1. A partir do momento em que uma arquitetura é definida e uma implementação suficientemente estável da plataforma virtual é obtida, a Fase 2 pode ser iniciada.

Na Fase 2 são desenvolvidos e verificados de forma incremental cada módulo ou função do sistema. Idealmente seria necessário um mecanismo para

realizar o isolamento de um erro entre plataforma virtual e o *software* sendo simulado. Pode-se utilizar mecanismos como apresentado por Cordeiro et al. (2009), porém nem sempre é possível realizar o isolamento, especialmente se o sistema for complexo. O momento de seguir para Fase 3 é um critério a ser definido de forma heurística. Em tese, quando o sistema estiver se encaminhando para seu formato final, em termos de desenvolvimento, e quando a maior quantidade de código possível tiver sido estressada o suficiente (seguindo os critérios e exigência definidas para o projeto), pode-se seguir para a Fase 3.

Finalmente, na Fase 3, são realizados os testes e validações necessárias a nível de sistema. Principalmente nessa etapa, possíveis falhas poderiam surgir durante as execuções, considerando erros provenientes não somente do *software* embarcado, mas também da plataforma virtual. Quando detectados e isolados, é necessário realizar realimentações e correções de volta a Fase 2 de projeto.

Grande parte desse fluxograma, principalmente as transições, é baseado em heurísticas. Não é de preocupação deste trabalho propor ou aplicar métodos de cobertura ou quantificação da verificação. Somente demonstrar aplicações e o potencial dessa metodologia dentro do contexto elaborado anteriormente.

4.3 Descrição Detalhada da Fase 1

Como pode ser observado na Figura 9, de forma esquemática, o desenvolvimento da plataforma virtual é relativamente simplificado. A criação dos modelos dos periféricos em TLM pode ser executado ao mesmo tempo que a criação do simulador. O preprocessador do ArchC garante a geração de um simulador com interface compatível com protocolos TLM implementados internamente. Dessa forma, do ponto de vista dos modelos dos periféricos, basta utilizar um mecanismo de herança de definições, realizadas nas bibliotecas disponibilizadas com ArchC, e implementar o comportamento interno desejado. Caso seja utilizado um modelo TLM gerado automaticamente, deve-se utilizar um *wrapper* para realizar a interface, pois não há garantias que os protocolos utilizados serão compatíveis. Por definição, modelagens utilizando SystemC/TLM, se conduzidas seguindo as orientações propostas pelo padrão, garantem interoperabilidade e reuso dos modelos (RIGO; AZEVEDO; SANTOS, 2011). Já foi mostrado em outros trabalhos que tal abordagem pode ser eficaz, mesmo em um curto período de tempo, especialmente considerando que muitos modelos já foram implementados[4] em TLM (ver (ZIJLSTRA, 2015)).

[4] Ou até mesmo empresas desenvolvedoras de IPs que fornecem modelos para simulação.

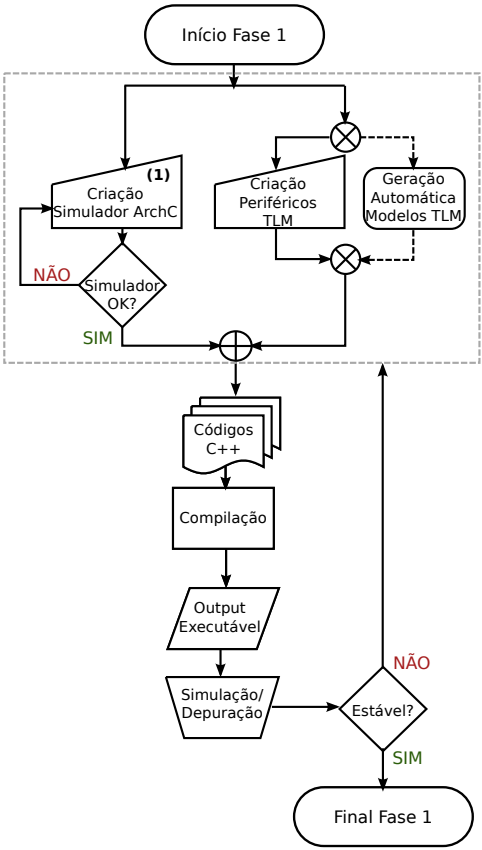

Figura 9 – Fluxograma representando as etapas internas da Fase 1.

A criação do simulador do conjunto de instruções é detalhada em forma de um fluxograma na Figura 10. Juntamente com o fluxo de desenvolvimento original proposto por Rigo, Azevedo e Santos (2011) uma pequena expansão, proposta neste trabalho, é mostrada no retângulo pontilhado.

No fluxo original da LDA ArchC, é necessário a criação de dois arquivos com descrições gerais da arquitetura e do conjunto de instruções. Um dos arquivos contém informações gerais como memória, registradores, ordenação dos bits (Figura 10 **(2)**). Enquanto no outro, são descritas informações relativas ao formato, tipo, campos e decodificação de cada instrução (Figura 10 **(1)**). A partir desses arquivos o preprocessador (um compilador em que o *backend* traduz as informações de entrada em um modelo em SystemC) gera um conjunto de protótipos que devem ser preenchidos, com o comportamento

de cada instrução descrita em (Figura 10 **(1)**). Também são gerados arquivos auxiliares, como *Makefiles* e arquivos para testes e execução do simulador. Como opção, pode-se criar internamente ao modelo um servidor de depuração (i.e. *GNU Debugger* (gdb)), utilizado para acessar a memória e registradores do simulador, e assim, depurar a aplicação que está sendo desenvolvida.

O processo de verificação que garante o comportamento correto do simulador, como proposto por Rigo, Azevedo e Santos (2011), é realizado através da execução de uma série de programas, com objetivo de estimular instruções específicas, definidos no *benchmark acStone*. Para cada programa, é esperado um conjunto de respostas específicas, comparadas com as respostas fornecidas pelo simulador através da interface `gdb`. Se divergências são encontradas, é possível isolar as instruções causadoras do erro. Rigo, Azevedo e Santos (2011) recomendam, após a execução satisfatória do *acStone*, a execução de *benchmarks* voltados para sistemas embarcados como Mibench[5], Mediabench[6] e SPEC2000[7]. Com isso, no final desse processo, é muito provável que se tenha um simulador suficientemente estável para suportar desenvolvimento e depuração de *software* embarcado.

Como alternativa, a esse processo manual, é proposto utilizar especificações em LTL para cada instrução. O formato $G(\varphi_1 \Rightarrow F\varphi_2)$ (em que φ_1 seria uma instrução e φ_2 os resultados esperados) é utilizado. A partir da escrita do comportamento esperado para cada instrução são criados monitores, por uma versão modificada (para gerar código SystemC/C++) da ferramenta `ltl2ba`. Posteriormente, esses monitores são inseridos de forma automática no código do simulador e são avaliados durante a execução. Caso haja alguma falha, os valores são impressos para o usuário. Dessa maneira, os monitores podem auxiliar na depuração do simulador e contribuir para obter uma versão estável mais rapidamente. Pode-se questionar a necessidade de se criar tais monitores, considerando a existência de construtores como `assert` em C++. Contudo, o objetivo dessa abordagem é no sentido de mostrar como mecanismos simples podem ser utilizados para depuração, sem agregar *overhead* no desenvolvimento. Em teoria, `assert` poderia ser utilizado, simplesmente criando pré e pós condições na execução de cada instrução.

4.4 Descrição Detalhada da Fase 2

A Fase 2, como detalhado na Figura 11, é divida em 3 partes principais. A etapa **(2a)** é dedicada ao desenvolvimento do *software* embarcado fazendo uso

[5] <http://wwweb.eecs.umich.edu/mibench/>
[6] <http://euler.slu.edu/~fritts/mediabench/>
[7] <http://www.spec2000.com/>

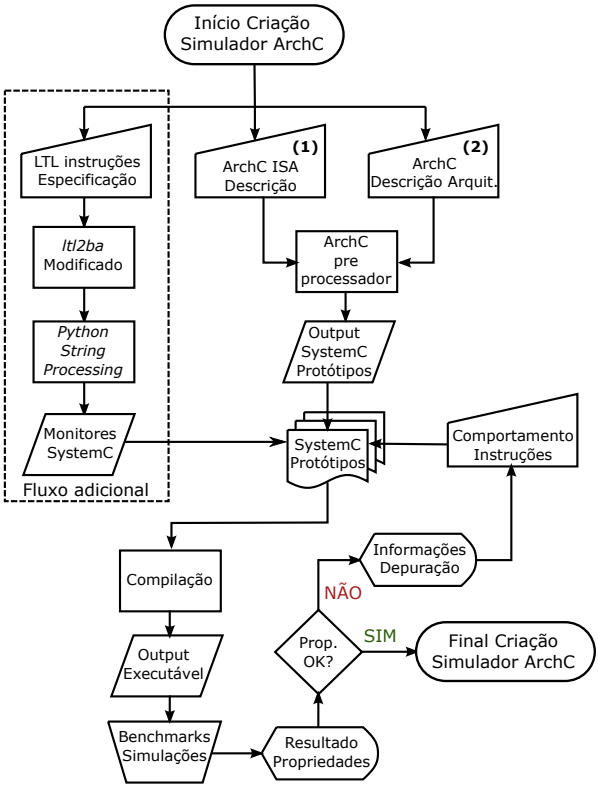

Figura 10 – Fluxograma representando as etapas internas da implementação de verificação de modelos com ArchC. Conforme mostrado em Figura 9 **(1)**. Região destacada pelo retângulo pontilhado representada fluxo adicional desenvolvido para auxiliar na verificação de modelos criados através de ArchC.

de metodologias como MBD, também com os benefícios de utilizar a plataforma virtual, criada na etapa anterior para depuração, por exemplo. Essa etapa é relativamente direta e, à medida que se obtém determinada funcionalidade, pode-se iniciar a segunda etapa **(2b)**.

Na etapa **(2b)**, é proposto um processo de exploração e procura de erros de implementação, baseado essencialmente nas capacidades do *model checker* ESBMC. Para lidar com complexidade em fases iniciais de projeto e considerar códigos parcialmente completos, a exploração por erros é feita com base no Grafo Estático de Chamada de Função (GSCF). A ideia principal é extrair o GSCF, diretamente do código, e percorrê-lo (a partir das folhas) até que todas

as funções tenham sido processadas pelo ESBMC[8]. Esse processo é realizado automaticamente e dois resultados são possíveis, como pode ser analisado no fluxograma (ver Figura 11). Caso o ESBMC não consiga verificar corretamente a função, por atingir o limite utilizado de memória ou tempo, o grafo é marcado, para que possa ser manualmente processado posteriormente. Uma forma de lidar com essa situação é obter das simulações valores que possam reduzir o espaço de estados (e.g., limite de laços ou valores para variáveis). Nos casos em que não ocorre um estouro dos limites, a função pode apresentar erros, ou não. Se erros foram encontrados, espera-se um contraexemplo, que deve auxiliar na depuração e correção na etapa **(2a)**. Se não foram encontrados erros, o grafo continua a ser percorrido até atingir a função `main` (ou o ponto de entrada definido no *script* que gera o grafo e executa o ESBMC); nesse ponto, pode-se passar para a etapa de verificação funcional **2c**. Essa abordagem foi proposta por Behrend et al. (2015) e reimplementada nesse trabalho seguindo os mesmos princípios.

A implementação dessa etapa é realizada através de um *script* na linguagem *Python*. Essa implementação processa os resultados gerados pela ferramenta cflow[9], gerando o GSCF e executando o ESBMC para cada um dos nós do grafo.

Finalmente, na etapa de verificação funcional (**2c**) é observado o comportamento desejado com base nas especificações para o sistema. O fluxograma de verificação segue como o base o trabalho realizado por Morse (2015). Inicialmente, são obtidas propriedades LTL das especificações. O ideal é que as especificações sejam formalizadas em algum formato, o qual pode favorecer a extração das propriedades. É claro que se pode especificar o sistema diretamente em LTL, mas essa não é uma prática comum. Após a definição das propriedades, deve-se criar testes (*harness* como definido originalmente pelo autor) que possuem o formato geral mostrado na Seção 5.2.2.3. Esses testes criam uma `pthread`, utilizada para executar simbolicamente o monitor e o programa internamente ao ESBMC, além de chamar a função que possui o comportamento observado. Os resultados podem ser dependentes de certos parâmetros de entrada, por parte do ESBMC (como explicado na Seção 3.1.2), dessa forma, experimentações podem ser necessárias.

Evidentemente que à medida que o sistema se torna mais complexo, a verificação é mais favorecida por simulações do que *Model Checking*. É claro que em tais cenários, não há garantias da ausência de erros, mesmo considerando

[8] Uma nota prática é a necessidade de instrumentar o código e alocar ponteiros, quando colocando um ponto de entrada em uma função, por exemplo; caso contrário ponteiros não existirão ou serão `NULL` para o ESBMC.

[9] <http://www.gnu.org/software/cflow/>

BMC, que não é um método completo. Apesar disso, certas metodologias (como apresentado aqui) têm o potencial de reduzir a probabilidade de um produto apresentar uma falha por um erro de programação. Justamente por introduzirem redundâncias e alternativas para estressar o sistema em diferentes casos específicos.

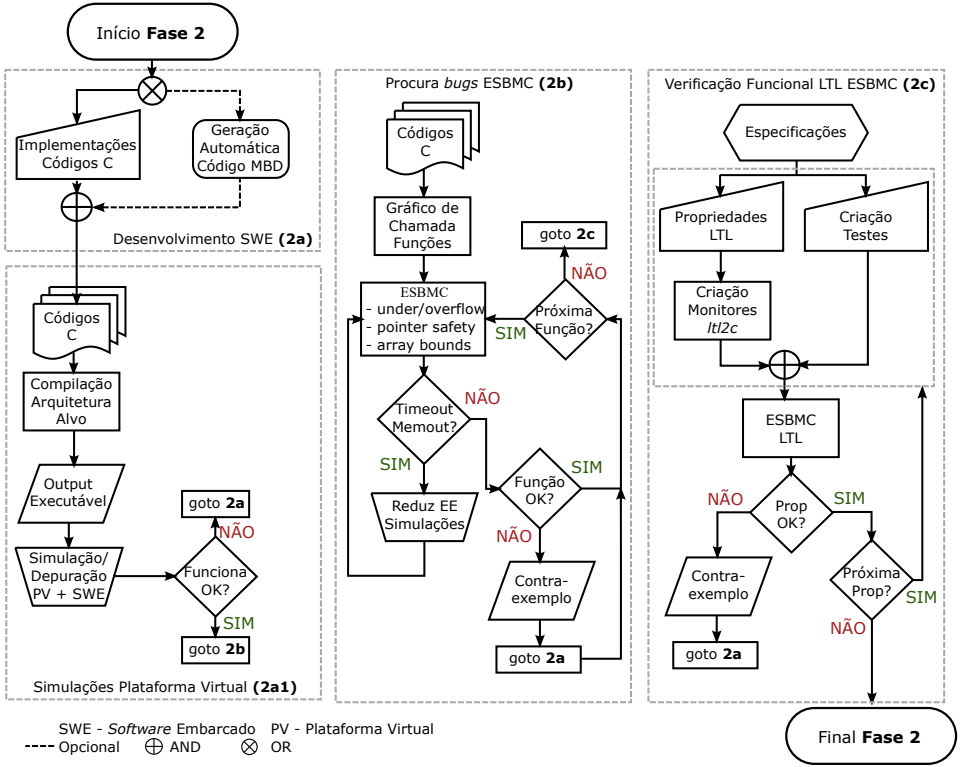

Figura 11 – Detalhes da Fase 2, desenvolvimento e verificação incremental.

4.5 Descrição Detalhada da Fase 3

Nessa etapa, são realizadas simulações completas, considerando todos os elementos que compõem o sistema. Propriedades que não foram verificadas com sucesso através do ESBMC podem ser utilizadas para monitorar a execução durante essa etapa.

Como o *software* embarcado está sendo executado em um ambiente virtual, há acesso a todas as variáveis e registradores de cada um dos processadores sendo simulados. Para estimular corretamente esse sistema, é necessário a criação de um *testbench* e certas informações de entrada.

Primeiramente, é necessário identificar variáveis que são vistas como entrada para determinada função, as quais não são modificadas dentro do escopo da função considerada; ou seja, são variáveis de somente leitura. Isso pode ser realizado manualmente, se o sistema não for muito complexo, ou utilizando alguma ferramenta de análise.

Após a identificação de possíveis alvos de estímulo é inevitável: conhecimento do momento certo para aplicar esse estímulo (1); saber em qual região da memória, ou em outras palavras o endereço, em que o valor deve ser escrito (2). É conhecido em tempo de compilação os endereços de valores globais e de funções. Além disso, através de *disassemblers* é possível obter o código *assembly* gerado, facilitando a identificação de contadores de programas relacionados a endereço de *return* de funções.

Com relação a (1), pode-se utilizar os endereços de entrada e saída de funções para saber exatamente o momento exato de amostrar e aplicar o estímulo através da interface do *testbench*. Soluções para (2) são mais diretas, pois se a variável for global, seu endereço já é conhecido em tempo de compilação. Caso contrário, é necessário instrumentar o código para obter endereços em tempo de execução de variáveis locais (e.g., utilizando um esquema ponteiro global recebe o endereço de variável local). Se há necessidade de mecanismos mais complexos para sincronização, modificações no *software* sendo verificado podem se tornar necessárias. Porém, tais abordagens seriam intrusivas, o que nem sempre é desejado nesse tipo de verificação.

Com essas informações, é ainda necessário definir os valores que servirão de estímulos. Nesse momento, não é focado em técnicas mais avançadas para realizar esse processo. São utilizados somente métodos simples para gerar estímulos randômicos restritos.

Grande parte dessas etapas podem ser teoricamente automatizadas através de alguns instrumentos específicos (e.g., *Binary File Descriptor library (BFD)*). Devido a certas particularidades, como será visto em algumas implementações demonstradas no capítulo seguinte, esse processo de automatização

pode ser relativamente complicado e preferiu-se por uma abordagem manual em primeira instância. De forma geral, essa etapa pode ser comparada com verificação de *hardware*, porém sem as mesmas restrições e detalhamento de tempo.

4.6 Conclusões e Discussão

Neste capítulo, foram descritas as três fases principais que compõem a metodologia utilizada no capítulo seguinte desse trabalho.

Na primeira fase, são desenvolvidos os modelos que irão compor a plataforma virtual, utilizada posteriormente para simulação e validação do *software* embarcado. Em comparação com fluxos tradicionais de desenvolvimento, essa etapa é adicional, ou seja, tem função exclusiva, voltada para metodologia ESL com plataformas virtuais. Além disso, é considerado a utilização de uma LDA para criação de simuladores já compatíveis e com interfaces para conexão com modelos TLM. Além disso, a abordagem proposta também é compatível com modelos já existentes (e.g., na indústria) ou em provedores como OpenCores e Imperas. Nestes casos, o custo (e tempo) de desenvolvimento seria eliminado ou extremamente reduzido (devido a possíveis adaptações).

A segunda fase é dedicada para criação e verificação incremental do *software*, com possibilidade para integração com ferramentas de geração automática de código, como será demonstrado posteriormente. Nessa fase, é inserido uma etapa de exploração por erros de implementação típicos, com a ferramenta ESBMC. Além disso, é possível avaliar o comportamento de forma modular utilizando as capacidades do ESBMC e também considerando a possibilidade de monitorar o comportamento em tempo de execução.

Por fim, são consideradas, na terceira fase, as propriedades e comportamento do sistema todo. As execuções podem ser monitoradas para avaliação de propriedades específicas e testes diretos. Para tal propósito, a plataforma virtual contribui aumentando a observabilidade dos estados do *software* embarcado, e assim, fornecendo mecanismos que dificilmente seriam possíveis para depuração e verificação do sistema em outros fluxos.

5 IMPLEMENTAÇÕES E RESULTADOS

Neste capítulo são relacionados os resultados mais relevantes do trabalho. Duas partes principais compõem este capítulo. A primeira é destinada a descrever os estudos de caso. Na segunda parte são demonstrados casos específicos de utilização da metodologia descrita no Capítulo 4.

A ideia é demonstrar diversos cenários de combinação entre *Model Checking*, simulação e desenvolvimento dentro do contexto das implementações e da metodologia proposta.

5.1 Descrição dos Estudos de Caso

O grande número de pesquisas recentes em verificação de *software* embarcado incentivou a elaboração de diversos *benchmarks* para comparação entre ferramentas diferentes. Nesse sentido, a competição SV-COMP tem papel crucial, em determinar um conjunto de *benchmarks* de forma a padronizar os estudos de casos, e assim, fornecer parâmetros quantitativos para comparações. Apesar disso, nem sempre é possível utilizar esses *benchmarks*, principalmente considerando domínios específicos de aplicação.

Neste trabalho são utilizados dois sistemas que foram implementados inicialmente com MBD e, posteriormente, integrados para verificação na metodologia utilizada (seguindo abordagens similares a propostas por Kroening et al. (2015)). O terceiro estudo de caso foi implementado diretamente na linguagem C utilizando o sistema operacional de tempo real `FreeRTOS`[1], para fornecer suporte a multitarefas e temporização. Para todos os estudos de caso é considerado o mesmo simulador de conjunto de instruções, também desenvolvido como parte deste trabalho, variando somente os periféricos e configurações da plataforma virtual utilizada.

5.1.1 Sistema de Controle de Injeção de Combustível (SCIC)

A grande maioria dos automóveis atuais utilizam sistemas baseados em *software* para controlar a mistura de combustível em motores a combustão. Como esses sistemas são baseados em algoritmos híbridos de controle (discreto e contínuo), o seu desenvolvimento é realizado em ferramentas destinadas para tal propósito, utilizando metodologias adequadas, i.e., Simulink® e MBD. Em fases posteriores de projeto, esses modelos em MBD são traduzidos para uma implementação em C ou C++, considerando não somente a arquitetura alvo, mas

[1] <http://www.freertos.org/>

também, a possibilidade de integração com sistemas operacionais de tempo real.

O Sistema de Controle de Injeção de Combustível[2] (SCIC) utilizado neste trabalho, é um modelo híbrido e tolerante a falhas implementado na ferramenta Simulink®, largamente conhecido e utilizado em outros trabalhos nessa área (ver Paiva et al. (2008)). Esse modelo representa uma grande variedade de sistemas que possuem as mesmas características. Complexidade relativamente baixa em termos de linhas de código (1284 LoC), porém com diversidade de recursos diferentes: aritmética, L.U.Ts (*Look-up Tables*), controladores integrais, filtros e interpoladores. Além disso, máquinas de estado finitos controlam os diversos modos de operação do sistema, garantindo operação mesmo em condições de falha de sensores.

Em termos de entradas e saídas, esse sistema de controle observa os valores de quatro sensores: oxigênio (O2/EGO), velocidade (SPEED), *throttle*[3] (THROTTLE) e pressão (MAP). Cada sensor é mapeado a uma máquina de estados, que indica se os valores fornecidos por esses sensores está dentro de faixas toleradas como operação normal, ou se o sensor está fornecendo valores falhos. Na Figura 12 é mostrado as máquinas de estados que indicam os valores dos sensores; na Figura 13 a máquina de estados que observa o número de falhas.

Com base nos valores fornecidos por esses sensores, o sistema tem objetivo de manter a relação ar/combustível (saída) relativamente constante durante operação. Além disso, falhas isoladas de sensores são toleradas, via estimação do parâmetros através de memória ou interpolação, com penalização de uma mistura rica (com concentração maior do que o necessário de combustível). Caso mais de um sensor falhe, inevitavelmente o combustível deve ser desligado.

Na Figura 14, pode ser observado os modos principais de funcionamento do sistema, os estados, e suas relações de transição. Os modos de operação são agrupados quanto ao sistema, ou com relação à mistura de combustível. Quanto ao sistema, há quatro modos: *Funcionando*, *Emissões Baixas*, *Mistura Rica* e *Combustível OFF (Desligado)*; a mistura de combustível, por outro lado, pode tomar três modos *LOW*, *RICH* e *OFF*. Se um dos sensores falhar, o sistema passa a funcionar com uma mistura com excesso de combustível, se mais de um sensor falhar, o sistema é desligado. O mesmo ocorre em caso de *overspeed*, ou sobre velocidade.

[2] <http://www.mathworks.com/help/simulink/examples/modeling-a-fault-tolerant-fuel-control-system.html>

[3] Normalmente se utiliza a denominação posição da borboleta, ou regular de pressão em português. O termo em inglês é mantido aqui por fins de clareza.

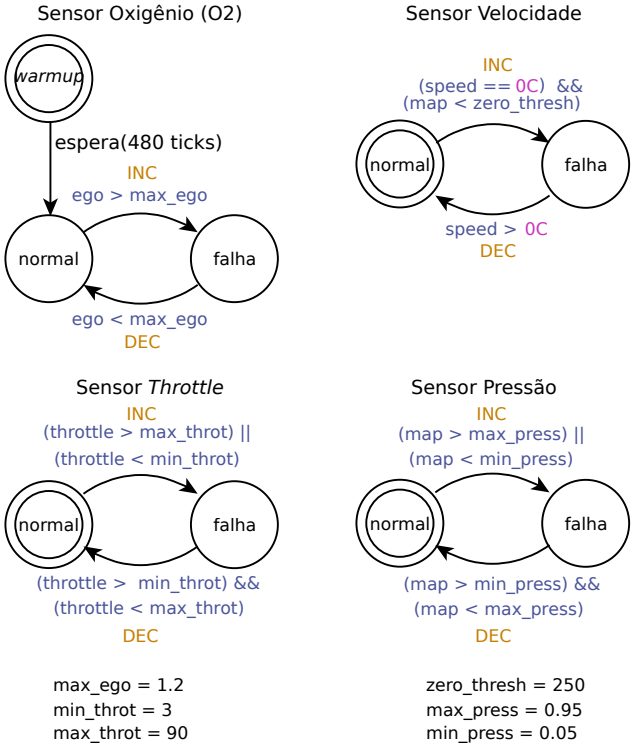

Figura 12 – Máquinas de estados que indicam os valores fornecidos pelos sensores (normal, falha). Valores mostrados na figura são como mostrados do ponto de vista de *software*, estão associados com grandezas físicas mas são adimensionais.

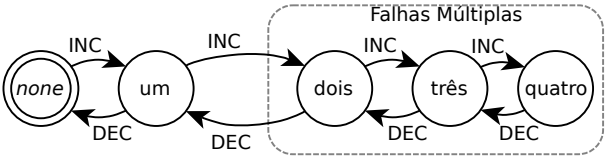

Figura 13 – Máquina de estados que indica o número de sensores em falha.

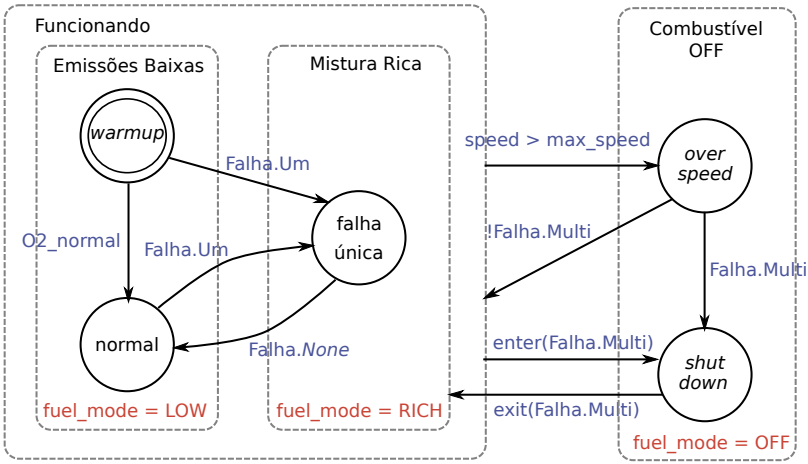

Figura 14 – Máquina de estados que controla os modos de operação do sistema: *funcionando*, *Emissões Baixas*, *Mistura Rica* e *Combustível OFF (Desligado)*.

Esse sistema funciona com base em um período de amostragem de 10 $milissegundos$, o qual é associado a um serviço de interrupção por *hardware* baseado essencialmente em tempo. Desta forma, as únicas alterações realizadas no código gerado automaticamente pela ferramenta, é associar a função que executa o algoritmo de controle com a interrupção. Fornecendo assim, a taxa de amostragem para o sistema. As configurações realizadas antes da geração do código garantem a compatibilidade com a arquitetura alvo, levando em conta tamanho da palavra principalmente.

Do ponto de vista de verificação, já foi demonstrado que o processo de geração de código pelo Simulink® é confiável e mantém as propriedades do sistema (STAATS; HEIMDAHL, 2008). Esse sistema em específico (SCIC) já foi verificado em pelo menos outro trabalho, de conhecimento do autor Paiva et al. (2008). Nesse trabalho, foi utilizado uma abordagem diferenciada para BMC, chamada de *Incremental BMC* (cosiderando *Model Checking* baseado em Satisfatibilidade Booleana) com foco principal em propriedades gerais do sistema, como a capacidade em manter constante a mistura ar/combustível. Apesar disso, não foram verificadas propriedades mais específicas ou estruturais.

Neste trabalho, os testes realizados com esse sistema focam justamente em erros típicos encontrados (propriedades de segurança) e também, propriedades específicas, que cobrem principalmente as transições nas máquinas de estado que governam o sistema.

5.1.2 Sistema de Controle de Acelerador Eletrônico (SCAE)

Seguindo na mesma área automotiva, esse segundo estudo de caso implementa um controlador de um acelerador eletrônico, também realizado na ferramenta Simulink®. O objetivo da utilização desse sistema é mostrar um esquema mais complexo com relação a Fase 1 da metodologia mostrada no capítulo anterior, e assim, estimular cenários específicos de desenvolvimento da plataforma virtual. Ao invés de focar na verificação do sistema, é explorado, através desse exemplo, formas de simulação similares a conceitos envolvendo *Software-in-the-Loop* e *Processor-in-the-Loop*, considerando um ambiente totalmente virtual.

Do ponto de vista funcional, esse sistema controla o posicionamento angular do mecanismo mecânico denominado *throttle*, também presente no estudo de caso anterior. Nesse estudo de caso, no entanto, é mostrado o controlador que fornece o valor de referência para o mecanismo, ao invés de um valor proveniente de um sensor. Para que as fins de clareza e simplicidade, a descrição do sistema pode ser abstraída do meio físico e relacionada somente em termos de entradas/saídas e funcionalidade.

O sistema possui duas entradas, uma de referência angular (em radianos) e outra de realimentação, proveniente de um sensor. Como saída, o controlador fornece uma referência para o atuador, também em radianos. O modelo desenvolvido com MBD possui um mecanismo de *safety* através de redundância. A ideia principal é utilizar *hardware* separados (dois processadores executam a mesma funcionalidade) para fazer amostragem e cálculo do sinal de controle, além de um árbitro para selecionar qual dos sinais será aplicado ao atuador. A arquitetura típica, nesse tipo de sistema (*drive-by-wire*), normalmente contém uma unidade de processamento central, em que é implementado o árbitro e redundância para os sensores, atuadores e processadores que calculam o sinal de controle (ISERMANN; SCHWARZ; STÖLZL, 2002). A comunicação entre esses módulos do sistema pode utilizar barramentos de comunicação CAN, ou outros, que forneçam confiabilidade e determinismos necessários.

Da mesma forma que o estudo de caso anterior, é gerado o código do algoritmo de controle e do árbitro, os quais são integrados em uma plataforma virtual baseada no microcontrolador MSP430. Além disso, o modelo do árbitro é encapsulado através de TLM e inserido na plataforma como um elemento implementado em *hardware* (uma espécie de periférico), o qual possui acesso direto as memórias dos dois microcontroladores que calculam o sinal de controle. O objetivo é mostrar as diferenças em se desenvolver tal sistema em MBD em comparação com ESL. Do ponto de vista de teste e validação, nem sempre, pode-se considerar entradas aleatórias como estímulos. Pensando

nisso, é utilizado o modelo do sistema mecânico, convertido em C, para fechar a malha de realimentação, fornecendo entradas significativas para o sistema.

Em suma, esse estudo de caso deve servir como forma de demonstração das diferenças entre as metodologias de desenvolvimento, vantagens e desvantagens. Aliás, é claro, de mostrar esquemas mais complexos de simulação considerando modelos físicos interagindo com o *software* em verificação.

5.1.3 Computador de Bordo de um *CubeSat*

5.1.3.1 Conceitos Gerais *CubeSats*: *Projeto FloripaSAT*

CubeSat é um formato de satélite em que as dimensões são reduzidas (formato cúbico de aresta de 10 cm), destinado principalmente para projetos acadêmicos de pesquisa. Esses pequenos satélites são divididos estruturalmente em módulos, cada um responsável por atividades específicas. Tipicamente os principais módulos necessários em tais satélites são: *On Board Data Handling* (OBDH), *Telemetry, Tracking and Command* (TTC) e *Electrical Power System* (EPS). As funções de cada módulo podem ser resumidas como:

OBDH: Controle e comando dos módulos e processamento de informações;

EPS: Gerenciamento energético, em alguns casos divide funções com o OBDH;

TTC: Envio e recebimento de informação à estação terrestre.

Normalmente, os *CubeSats* transportam, além dos seus módulos, uma carga útil relativamente simples,[4] com aplicações variadas, desde experimentos biológicos até observação ou monitoramento atmosférico.

As informações mais relevantes para o contexto explorado neste trabalho dizem respeito a arquitetura utilizada no projeto FloriSAT.[5] Nesse *CubeSat*, será utilizado o microcontrolador MSP430 da Texas Instruments em todos os seus módulos (OBDH, EPS e TTC). A troca de informações entre os módulos será realizada através de um barramento de comunicação I2C, devido a simplicidade e utilização prévia em outros projetos do gênero. Um diagrama esquemático do FloripaSAT é mostrado na Figura 15.

Com base nessas definições gerais, Zijlstra (2015) criou uma plataforma virtual refletindo as características estruturais principais do projeto FloripaSAT. As implementações realizadas por Zijlstra (2015) focam principalmente na modelagem em transações (TLM) dos periféricos de temporização (*Timed Interruptions*) e de comunicação I2C. A Figura 16 representa um esquemático da plataforma virtual montada para o FloripaSAT. Essa plataforma fornece

[4] Considerando os formatos padrões 1U, uma unidade cúbica.

[5] Uma iniciativa de professores e alunos na Universidade Federal de Santa Catarina para desenvolvimento e lançamento de um *CubeSat* 1U.

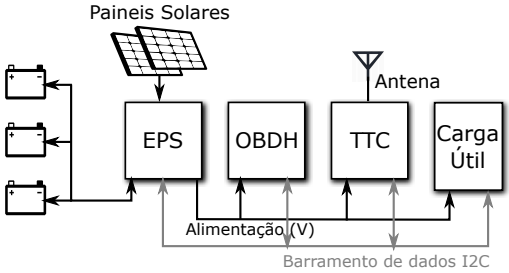

Figura 15 – Diagrama esquemático mostrando os módulos do FloripaSAT. Adaptado de (VILLA et al., 2014).

suporte ao desenvolvimento e verificação antecipada do *software* embarcado, dentro do contexto exposto na Fase 1 da Figura 7.

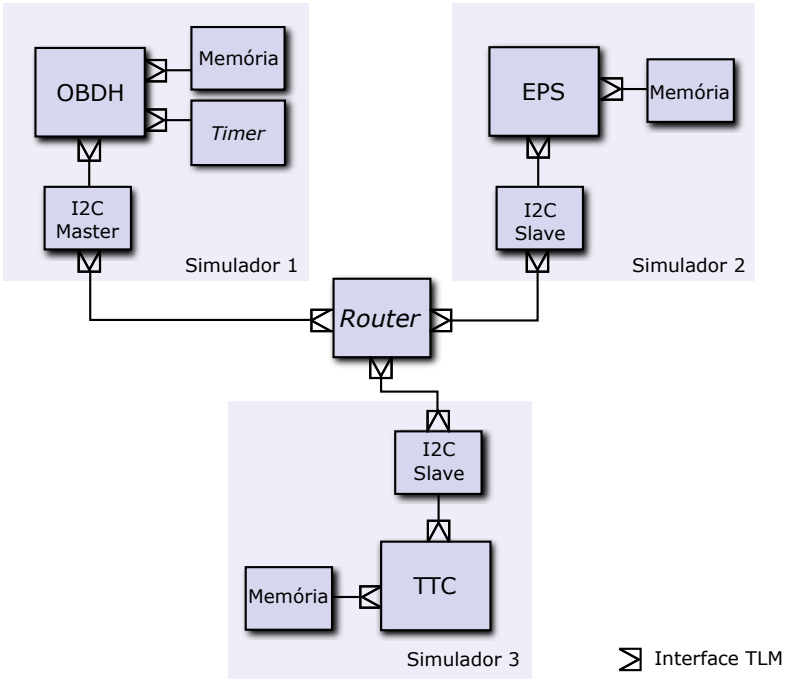

Figura 16 – Diagrama da Plataforma Virtual do FloripaSAT. Cada simulador é uma instância de um emulador do conjunto de instruções do microcontrolador MSP430, criado com ArchC. Adaptado de (ZIJLSTRA, 2015).

Uma particularidade comum aos *CubeSats*, que define limitações ao

projeto e operação em órbita, refere-se as restrições energéticas. As dimensões reduzidas implicam em uma menor área para instalação de painéis solares, principal[6] fonte de energia para maioria dos *CubeSats*. Como consequência, as maiores preocupações são na redução do consumo e aumento da vida útil das baterias, tipicamente através de rotinas implementadas em *software*.

Dois fatores são relevantes quanto ao consumo de energia e devem ser considerados durante projeto do sistema. O primeiro diz respeito ao TTC, que é tipicamente o módulo que mais consome energia, devido necessidade em amplificação do sinal de comunicação. Outro fator importante é que a Estação Terrestre (ETR) só tem visibilidade de comunicação durante uma janela da ordem de alguns minutos. Levando em conta essas afirmações, é possível desligar o módulo TTC, quando não há visibilidade da ETR, e assim, reduzir o consumo de energia.

Tipicamente, as altitudes orbitais de *CubeSats* são da ordem de 200 km a 1000 km. Isso implica em períodos orbitais (tempo necessário para completar uma revolução em torno da Terra) próximos de 88 a 105 minutos. Tomando como exemplo o *CubeSat* NanoSat-BR1[7], sua altitude orbital de lançamento foi em torno de 630 km, determinando um período de aproximadamente 96,7 minutos[8].

Em resumo, essas informações delimitam um cenário para criação de uma rotina em *software*, em que o OBDH controlaria o estado (ligado/desligado) do módulo de comunicação (TTC), com base nas informações de visibilidade da ETR. Além disso, é necessário realizar o monitoramento do estado da bateria, bem como outras tarefas que podem ser necessárias.

5.1.3.2 Descrição e Especificação do Estudo de Caso: *Computador de bordo FloripaSAT*

Com base nas informações fornecidas na seção anterior, foi criado um exemplo de computador de bordo, que considera o período orbital[9] do NanoSat-BR1, e realiza o desligamento do módulo TTC. Além disso, uma segunda tarefa recebe o *status* da bateria, proveniente do módulo EPS.

Os estados de funcionamento do OBDH são mostrados na Figura 17. É assumido que o módulo TTC irá fornecer o *status* da ETR, o mecanismo pelo

[6] A irradiância (W/m^2) dos painéis pode ser de 3 a 6 vezes maior que a irradiância por Albedo terrestre ou radiação infravermelha (Aalborg University's Student Satellite, 2002).

[7] Projeto realizado pela Universidade Federal de Santa Maria.

[8] Calculado utilizando Terceira Lei de Kepler.

[9] Uma escala é considerada para facilitar a simulação. Os 96.7 minutos (≈5800 segundos) são simulados como 58 milissegundos.

qual essa informação é obtida não é de preocupação nesse momento. O estado S1 representa um modo de inicialização do sistema, para lidar com situações transientes. Como por exemplo, quando é realizado o lançamento há um tempo para estabilização, ou em situações de reinicialização (como um mecanismo de *watchdog*). A partir do momento que a ETR é visível, o OBDH passa para o estado S2. Nesse estado é monitorado o *status* da estação terrestre, obtido do TTC via I2C, e quando não há visibilidade a comunicação é desligada (estado S3). No estado S3, o módulo TTC é mantido desligado até o período orbital ser completado.

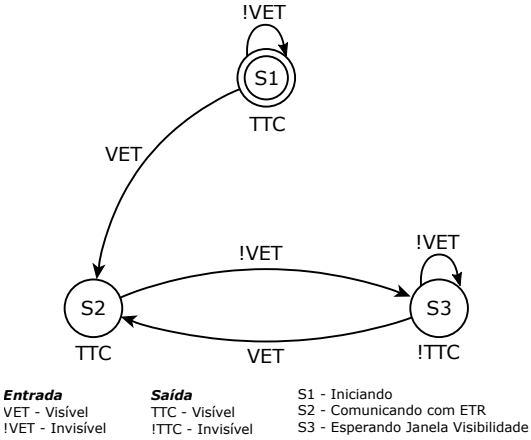

Figura 17 – Estados de funcionamento do OBDH. Informações são obtidas com base nas informações recebidas via I2C.

Para descrição das tarefas e sequências de execução do OBDH, foram utilizados diagramas de sequência. Duas tarefas principais são implementadas utilizando o FreeRTOS, simbolizadas por *Tarefa EPS* e *Tarefa TTC* na Figura 18. Além disso, um protocolo simples de comunicação é estabelecido entre o OBDH e o TTC, de forma a possibilitar o envio e recebimento de diversos comandos. Basicamente são utilizados 4 valores decimais: `ask_pwr_status` (44), `set_ttc_on` (55), `set_ttc_off` (66), `ask_ground_status` (77).

Para observar as sequências de execução, é preciso descrever os modos de funcionamento e configuração do periférico I2C e do microcontrolador. O MSP430 possui modos de funcionamento (*low power*), em que é possível desligar a unidade de processamento (CPU), de forma a reduzir o consumo de energia. Esse modo de funcionamento é modelado no simulador em SystemC e representa o mecanismo pelo qual o módulo TTC é desligado. O retorno desse modo de operação é realizado em uma *Interrupt Service Routine* (ISR),

acionada via interrupção externa, nesse caso quando o bit de início de uma transmissão I2C é recebido. Além disso, o modo em que os periféricos I2C são configurados (OBDH como *Master*, outros módulos com *Slaves*, ver Figura 16) exigem que as comunicações sejam sempre iniciadas pelo OBDH. Levando em consideração essas informações, para o OBDH obter uma resposta do TTC é necessário primeiramente enviar o respectivo comando (`ask_ground_status`), e depois, solicitar novamente a resposta (`get_ground_status`). No caso do EPS, não é necessário realizar essa comunicação em duas etapas, pois o EPS sempre responde com a mesma informação. Finalmente, considerando a Figura 18, a cada 8 milissegundos o OBDH completa um ciclo de funcionamento, executando duas vezes a *Tarefa EPS* e uma vez a *Tarefa TTC*.

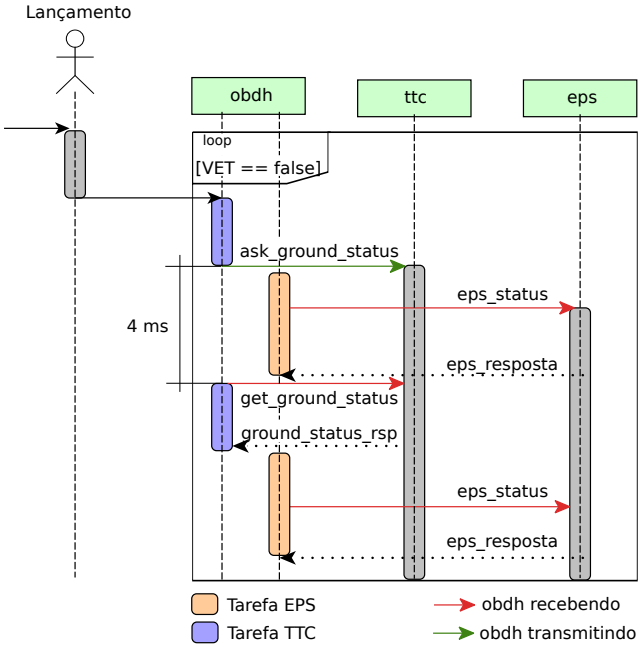

Figura 18 – Diagrama de sequência para as tarefas realizadas no estado S1.

Na Figura 19, é mostrado o diagrama de sequência para o estado S2. Nesse estado, somente é considerado trocas de informação com relação ao *status* da ETR. É importante perceber que este estudo de caso, embora válido para ser utilizado como prova de conceito neste trabalho, não tem por objetivo representar fielmente o computador de bordo que será utilizado no projeto FloripaSat. Isso fica claro nas tarefas sendo mostradas na Figura 19, pois a aplicação final vai processar e enviar um número consideravelmente maior de

informações. Para mostrar possíveis variações entre as sequências de execução na Figura 19, a *Tarefa TTC* é suspensa por 8 milissegundos.

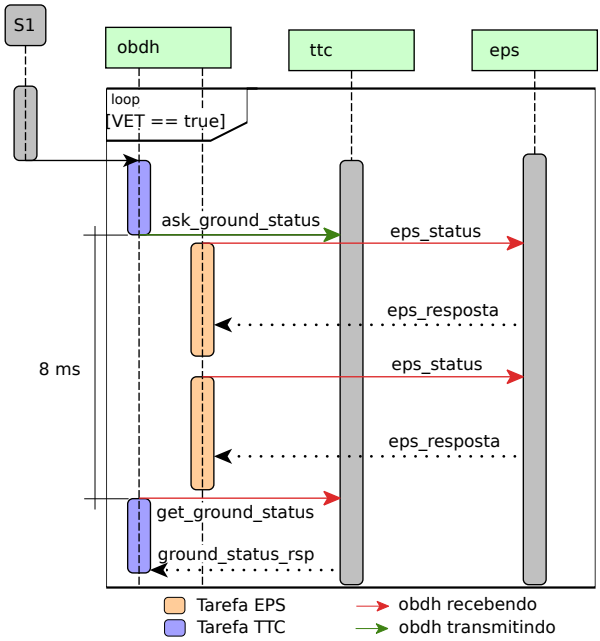

Figura 19 – Diagrama de sequência para as tarefas realizadas no estado S2.

Por fim a sequência de execução mais crítica é com relação ao estado S3, em que o TTC é desligado. A Figura 20 descreve detalhadamente a execução nesse estado. A primeira atividade do estado S3 é desligar o TTC, suspendendo a execução dessa tarefa por metade do período orbital, 29 milissegundos na escala de tempo utilizada para a simulação.[10] Não há nenhuma razão específica para que o TTC seja ligado nesse momento, além é claro de fornecer um *status* quanto a visibilidade da ETR e introduzir variações no diagrama de sequência. Não é esperado que a ETR esteja visível antes de pelo menos 58 milissegundos; dessa forma, segundo o comportamento esperado, o TTC deve manter-se desligado.

Esses três diagramas de sequência representam um computador de bordo simplificado, baseado na arquitetura do FloripaSat, que leva em consideração dados reais de outros *CubeSats* como parâmetros de entrada para sua especificação. Um erro crucial introduzido, do ponto de vista funcional, nas

[10] O *tick* do sistema operacional é configurado via o periférico modelado em TLM para fornecer interrupções a cada 1 milissegundo.

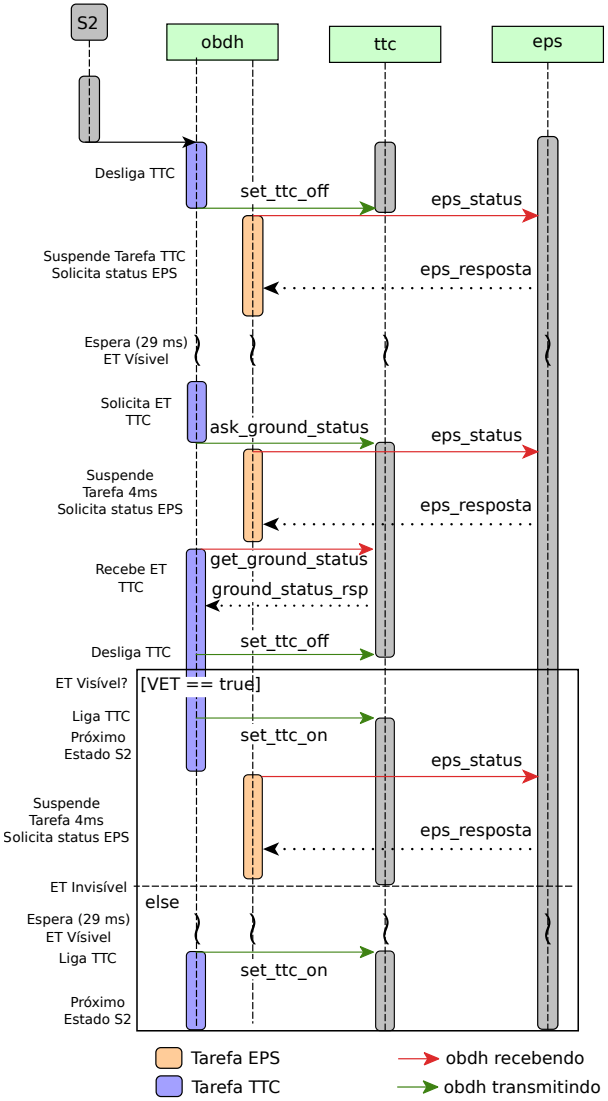

Figura 20 – Diagrama de sequência para as tarefas realizadas no estado S3.

especificações do sistema, que deve ser evidenciado posteriormente, é que o TTC não pode ser desligado baseando-se em um valor estático de tempo, por duas razões. Apesar de calculado o período orbital, muitas perturbações não consideradas[11] influenciam a velocidade; consequentemente, haverá divergências nesses valores. Outro fator importante é o decaimento da órbita, ou seja, uma vez lançado, com o tempo, a altitude vai reduzindo até o *CubeSat* ser destruído. É evidente que o desligamento do TTC deve levar em conta valores dinâmicos, amostrados durante operação, além é claro de *status* de bateria e outros fatores não considerados. Mesmo assim, com essas simplificações, é possível demonstrar propriedades interessante e estimular especialmente a Fase 3 da metodologia (ver Figura 7).

5.2 Resultados

Assim como os estudos de caso, os resultados obtidos são divididos em três seções. O simulador do conjunto de instruções da família de microcontroladores MSP430 é exposto inicialmente, relacionado a Fase 1. Na segunda seção, são utilizados os sistemas criados com MBD, para verificação incremental com ESBMC e simulação (Fases 2 e 3). Por fim é mostrado o estudo de caso do *CubeSat*, um sistema consideravelmente mais complexo.

5.2.1 Modelagem do Microcontrolador MSP430 com ArchC

Os microcontroladores da família MSP430 possuem uma arquitetura RISC de 16 bits com 27 instruções[12] e 16 registradores. O modelo de memória usa endereçamento linear, uma arquitetura Von Neumann, com barramento de 16 ou 8 bits e no máximo 128 kB. Os registradores R0 a R3 são utilizados como *program counter*, *stack pointer*, *status register* e *constant generator*, respectivamente. Enquanto os registradores R4 a R15 são para propósitos gerais. A arquitetura é ortogonal, ou seja, com algumas exceções todas as instruções têm acesso a todos os 7 modos de endereçamento.

As instruções são classificadas segundo os manuais da Texas Instruments conforme número de operandos, *Double* ou *Single*, ou instruções *Jump*.

Double Operand: Instruções aritméticas ou lógicas. Exemplo `mov src, dst`, equivalente a `dst = src` em C;

[11] Para o calculo foi utilizado Terceira de Kepler, que determina um valor de período orbital considerando a órbita e a massa dos corpos celestes. No entanto, arrasto aerodinâmico, perturbações de outros corpos celestes ou até mesmo o próprio movimento do *CubeSat* pode influenciar, o que não foi previamente considerado.

[12] 27 físicas e 24 emuladas.

Single Operand: Manipulação de um único operando ou controle. Exemplo `push src`, "empurra" palavra ou byte no *Stack*;

Jump: "Salto" para o endereço de destino. Exemplo `jmp label`, *Program Counter* é atualizado com o *PC* + *offset*.

O tamanho das instruções podem variar entre 1 ou 3 palavras dependendo do modo de endereçamento utilizado. Juntamente com essas informações são considerados os formatos e codificação de cada instrução como entrada das descrições para o `acsim`, além do preprocessador[13] do ArchC que gera os protótipos do modelo.

Os arquivos mostrados nos Programas 3 e 4 representam as principais informações de entrada para o `acsim`. Os arquivos gerados são posteriormente preenchidos com o comportamento de cada instrução. Por questões de clareza os arquivos gerados são parcialmente apresentados no Apêndice A.

Programa 3 Descrições gerais da arquitetura do MSP430, considerando memória e portas de interrupção TLM (`msp430.ac`).

```
1   /// MSP430 descrição
2   AC_ARCH(msp430) {
3     /// Arquitetura informações gerais
4     ac_wordsize 16;          /**< Tamanho da palavra */
5     ac_tlm_port DM:128K;     /**< Porta TLM, MSP430x2 */
6     ac_regbank RB:16;        /**< Banco de registradores */
7     ac_tlm_intr_port inta;   /**< Porta de Interrupção */
8     ac_tlm_intr_port intb;   /**< Porta de Interrupção */
9     ac_tlm_intr_port intc;   /**< Porta de Interrupção */
10
11    /// ArchC construtor
12    ARCH_CTOR(msp430) {
13      ac_isa("msp430_isa.ac"); /**< Arquivo de descrição dos formatos */
14      set_endian("little");  /**< Ordenação */
15    };
16  };
```

O modelo final gerado, sem nenhuma interface TLM, possui cerca de 4362 linhas de código, das quais 382 foram utilizadas para descrição do modelo, e 2018 foram escritas na descrição do comportamento. O restante (1962 linhas de código) foram geradas automaticamente, sem contar as bibliotecas e cabeçalhos importados automaticamente pelo ArchC.

Esse modelo foi posteriormente expandido com os outros construtores que o ArchC fornece, como interfaces de depuração `gdb`, emulação de chamadas do sistema operacional e os periféricos modelados por Zijlstra (2015). Além disso, esse modelo foi estressado intensamente e executou um grande número de instruções durante desenvolvimento.

[13] Na realidade um compilador que traduz a LDA em protótipos em SystemC.

Programa 4 Declaração dos formatos das instruções e decodificação (`msp430_isa.ac`).

```
1  /// MSP430 descrição das instruções
2  AC_ISA(msp430) {
3
4    /// instruções Double operand
5    ac_format Double_48 = "%odst:16 %osrc:16 %op:4 %sreg:4 %ad:1 %bw:1 %as:2 %
         dreg:4"; /**< Double instruction 48bits size */
6
7    ...
8
9    /// instruções Single operand
10   ac_format Single_32 = "oset:16 %cnt:6 %op:3 %bw:1 %as:2 %sreg:4";       /**<
         Single Instruction 32bits size */
11
12   ...
13
14   /// instruções Jump
15   ac_format Jump = "%opc:3 %c:3 %offset:10:s";                            /**<
         Formato Único */
16
17   /*****************************************************************//**
18   * Double Operand
19   ********************************************************************/
20   /** Instruções são separadas nas várias codificações possíveis para
21   *  Cobrir todos os casos
22   *
23   */
24
25   /// 48 bits
26   ac_instr<Double_48> mov_48imm, mov_48idm,
27   add_48idm, add_48imm,
28   ...
29   /*****************************************************************//**
30   * instruções Single Operand
31   ********************************************************************/
32   /// 32 bits
33   ac_instr<Single_32> rrc_32sm, rrc_32am, rrc_32imm, rrc_32im,
34   rra_32sm, rra_32am, rra_32imm, rra_32im,
35   ...
36
37   /*****************************************************************//**
38   * Instruções Jump
39   ********************************************************************/
40   ac_instr<Jump> jne, jeq, jnc, jc, jn, jge, jl, jmp;
41
42   ...
43
44   /// Decodificação
45
46   ...
47
48   /*****************************************************************//**
49   * Jump
50   ********************************************************************/
51   /// - JNE/JNZ Instrução
52   jne.set_decoder(opc=1, c=0);
53   /// - JEQ/JZ Instrução
54   jeq.set_decoder(opc=1, c=1);
55   /// - JNC Instrução
56   jnc.set_decoder(opc=1, c=2);
57
58   ...
59 };
60 };
```

Com relação a desempenho de execução,[14] resultados mostram variações entre 56 MIPS (*Millions Of Instructions Per Second*) e 78 MIPS. Mesmo utilizando um grande número de instruções para emular somente uma instrução da arquitetura alvo, esses resultados mostram um ganho considerável com relação ao *hardware*, que apresenta desempenho em torno de 8 a 25 MIPS (INSTRUMENTS, 2016) (em torno de 224% de ganho na pior situação). É importante destacar que esses resultados foram obtidos considerando somente o simulador do conjunto de instruções juntamente com a estrutura de dados como memória (definida internamente pelo ArchC). Quando utilizando outros periféricos ou unidades de processamento, o desempenho é consideravelmente menor, devido ao *overhead* introduzido pelo *kernel* do SystemC.

5.2.1.1 Verificação de Modelos Gerados pelo ArchC

Originalmente, o fluxo de verificação dos modelos gerados pelo ArchC é baseado em processos manuais de depuração e execução incremental. Para fornecer uma alternativa para esse problema, foi utilizado um mecanismo relativamente simplificado de asserções, através de descrições em LTL.

Basicamente foram introduzidas modificações na parte de geração de código da ferramenta `ltl2c`[15], para criação de um monitor em SystemC. Esse monitor funciona seguindo o mesmo princípio das ferramentas comerciais que oferecem asserções em SystemC. Cada propriedade é encapsulada em uma classe que é ligada a um módulo em SystemC. Essa classe recebe no seu construtor um ponteiro para o módulo, que fornece acesso a todas as suas variáveis, eventos e sinais. Então, o algoritmo implementado na ferramenta original `ltl2ba`[16] cria um monitor baseado em autômatos Büchi, que em tempo de execução avalia a propriedade.

A execução de modelos criados em SystemC, que utilizam processos `SC_THREAD`, é determinada pela implementação interna do `kernel`. Assim, funções como escalonamento e mudanças de contexto não são visíveis para o desenvolvedor, que não governa como os processos são executados. A única maneira de controlar a execução é utilizando eventos que indicam se determinado processo está pronto para ser executado ou não. Do ponto de vista dos monitores, isso é um problema, pois não tem como garantir a execução do monitor dentro das restrições necessárias para determinar a validade de uma

[14] Pode variar considerando o código sendo executado. Todos os testes neste trabalho foram realizados em uma máquina virtual com sistema operacional Ubuntu, 12GB de memória, e 4 núcleos Intel® i7-4790 CPU @ 3.60GHz (Somente 1 é utilizado).

[15] <http://svlab.hussamaismail.eti.br/ltl2c.zip>

[16] <http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/>

propriedade. Como consequência, grande parte dos trabalhos[17] que utilizam monitores baseados em autômatos, introduzem mudanças significativas na implementação interna do SystemC, de forma a expor ou adicionar fases de execução e garantir avaliação correta dos monitores/propriedades. Está fora do escopo deste trabalho incorporar tais mudanças. Assim, as especificações são limitadas a um conjunto reduzido de propriedades possíveis. Em outras palavras, propriedades que podem ser traduzidas em autômatos Büchi com apenas dois estados (e.g. pergunta-resposta $G(\varphi_1 \Rightarrow F\varphi_2)$ e propriedades de segurança $G(\varphi)$).

Esses monitores foram aplicados para monitoramento da execução das instruções de dois simuladores (MSP430 e MIPS) criados com o ArchC. Os modelos são modificados automaticamente, usando *scripts* e expressões regulares em Python, sendo adequados para execução dos monitores. Essa abordagem é preferível, pois não introduz mudanças nas ferramentas distribuídas com ArchC.

As propriedades especificadas possuem formato similar ao mostrado a seguir:

```
G (strcmp(mm->ISA.get_name(), "mov_16rm") == 0) ⇒ F(mm->dreg == mm->sreg)
```

Nesse caso, `mm` é um ponteiro para instância do simulador `msp430`; `ISA` é um objeto da classe `msp430_isa`, que contém as definições do conjunto de instruções; o método `get_name()` retorna um `const char *`, comparado ao nome da instrução especificada `mov_16rm`; `dreg` e `sreg` são variáveis utilizadas temporariamente para armazenar os valores dos registradores. Essa instrução `mov_16rm` deve mover uma palavra do registrador de origem para o registrador de destino.

Os Programas 5 e 6 apresentam a implementações do monitor gerado para essa propriedade. O evento `instruction_executed` coloca a `SC_THREAD` como pronta para ser executada logo após uma instrução ser executada pelo modelo (Programa 5 linha 29).

Para cada vez que o monitor é executado, se a condição definida na macro `mov_16rm_cexpr_1` for falsa e `mov_16rm_cexpr_0` for verdadeira, a propriedade é dita como falsa. As variáveis declaradas no Programa 5 (linhas 18 e 19) são utilizadas como contadores do número de vezes que o monitor foi executado e se houve alguma falha. No fim das simulações, esses valores são reportados para o usuário. A macro `dbg_printf()` redireciona `stdout` para `stderr` e imprime os valores das variáveis no momento da falha.

[17] Ver (TABAKOV; ROZIER; VARDI, 2012; PIERRE et al., 2012; SILVA et al., 2014; ECKER et al., 2007; PIERRE; FERRO, 2008; Zhaorong Xiong; Jinian Bian; Yanni Zhao, 2010; DUTTA; TABAKOV; VARDI, 2014).

Programa 5 Cabeçalho de declaração do monitor (`monitor.h`).

```
1  /* Cabeçalho gerado automaticamente */
2  #ifndef mov_16rm_mon_H
3  #define mov_16rm_mon_H
4
5  #include "systemc.h"
6  #include "msp430.H"
7
8   typedef enum {
9    mov_16rm_state_0,
10   mov_16rm_state_1,
11 } mov_16rm_state;
12
13 class mov_16rm_mon: public sc_module {
14   public:
15   mov_16rm_state mov_16rm_statevar; // variáveis de estado
16   msp430 * mm; // ponteiro para modelo a ser monitorado
17
18   unsigned int mov_16rm_ex;
19   unsigned int mov_16rm_fail;
20
21   void behavior(); // comportamento do monitor
22
23   SC_HAS_PROCESS( mov_16rm_mon );
24
25   mov_16rm_mon(sc_module_name name_, msp430 &module) { // construtor
26      mov_16rm_statevar = mov_16rm_state_0; // estado inicial
27      mm = &module;
28      SC_THREAD( behavior );
29      sensitive << mm->instruction_executed;
30   }
31   virtual ~mov_16rm_mon();
32 };
33 #endif // mov_16rm_mon_H
```

Como forma de demonstração, foram especificadas propriedades para as instruções `mov`, em todos os modos de endereçamento considerados no modelo do MSP430 (8 no total). Para o modelo MIPS,[18] 10 propriedades assertam as instruções aritméticas. Como estímulo foi considerado os *benchmarks* típicos: `Mibench`[19] (BasicMath) e `SNURT`[20] (Completo). Os testes foram executados 10 vezes, com valores médios reportados na Tabela 3.

Os monitores introduzem um *overhead* de execução esperado, cerca de 20% (2,56% por monitor) para o pior caso (SNURT MSP430). Outros trabalhos apresentam abordagens mais robustas e com resultados melhores que os obtidos (e.g (TABAKOV; ROZIER; VARDI, 2012)). Apesar disso, a abordagem utilizada funciona como prova de conceito e através desses recursos foi possível capturar um erro de implementação (na descrição de uma instrução) no modelo do MSP430 que passou despercebido pelos métodos originais do ArchC.

[18] Esse modelo é distruído pelo ArchC Team e pode ser encontrador em <https://github.com/ArchC/mips>.

[19] <http://wwweb.eecs.umich.edu/mibench/>

[20] <http://www.cprover.org/goto-cc/examples/binaries/SNU-real-time-bin.tar.gz>

Programa 6 Arquivo de implementação do monitor (`monitor.cpp`).

```
1   /* Implementação do Monitor */
2   #include "mov_16rm_mon.h"
3   #include "ansi-colors.h"
4   #include "ac_debug_model.H"
5
6   #define mov_16rm_cexpr_0 (strcmp(mm->ISA.get_name(), mov_16rm ) == 0)
7   #define mov_16rm_cexpr_1 (mm->dreg_mov_16rm == mm->sreg_mov_16rm)
8
9   void mov_16rm_mon::behavior() {
10
11    while(1) {
12      switch (mov_16rm_statevar) {
13        case mov_16rm_state_0:
14        if (!mov_16rm_cexpr_1 && mov_16rm_cexpr_0) {
15          mov_16rm_statevar = mov_16rm_state_1;
16        }
17        break;
18
19        case mov_16rm_state_1:
20        if (!mov_16rm_cexpr_1) {
21          mov_16rm_statevar = mov_16rm_state_1;
22        }
23        break;
24
25        default:
26        mov_16rm_statevar = mov_16rm_state_0;
27        break;
28      } // switch (mov_16rm_statevar)
29
30      mov_16rm_ex++;
31
32      if (mov_16rm_statevar == mov_16rm_state_1) {
33        dbg_printf(CB_RED CF_WHITE "Property mov_16rm FAILED" C_RESET LF);
34        dbg_printf(CB_RED CF_WHITE "sreg: " C_RESET LF, mm->sreg);
35        dbg_printf(CB_RED CF_WHITE "dreg: " C_RESET LF, mm->dreg);
36        mov_16rm_statevar = mov_16rm_state_0; // restarts the monitor
37        mov_16rm_fail++;
38      } // if (mov_16rm_statevar == _mov_16rm_state_1)
39
40      mm->next_instruction.notify();
41
42      wait();
43    } // while(1)
44  }
45  mov_16rm_mon::~mov_16rm_mon () {};
```

Tabela 3 – Sumário dos resultados obtidos. Número total de instruções executadas (Σinst), número de vezes que os monitores foram executados (Σ*P*).

	Tempo (s)	Memória (MB)	Σinst	ΣP
	MIPS ArchC Monitorado			
SNURT	2,845	4,81	306142008	61668661
BasicMath	5,364	5,11	632938677	1229046810
	MIPS ArchC NÃO Monitorado			
SNURT	2,515	4,81	306142008	0
BasicMath	4,583	5,11	632938677	0
	MSP430 ArchC Monitorado			
SNURT	0,488	4,99	18777555	5234857
BasicMath	<0,01	4,95	36547	7011
	MSP430 ArchC NÃO Monitorado			
SNURT	0,405	5,00	18777555	0
BasicMath	<0,01	4,92	36547	0

5.2.2 Verificação do *SCIC*

5.2.2.1 Criação da Plataforma para Simulação

Os resultados obtidos na verificação do *SCIC* foram divididos em dois cenários diferentes. O primeiro é utilizando somente *Model Checking* para verificação de propriedades de segurança (limites de arranjos, divisão por zero, segurança de ponteiros e *overflow*). Mediante os resultados obtidos do *model checker* foram conduzidas simulações do sistema, considerando entradas aleatórias (não determinísticas). Com base nessas simulações foi possível determinar alguns valores limites, os quais foram posteriormente utilizados como entrada para o ESBMC. Essa segunda interação é realizada de forma manual e define o segundo cenário de verificação, abrangendo principalmente as Fases 2 e 3 da metodologia demonstrada no Capítulo 4.

Iniciando na Fase 1 da metodologia, é criado uma plataforma virtual do sistema, constituída de uma instância do simulador MSP430, uma memória TLM e um *timer*, para fornecer interrupções por tempo. O Programa 7 mostra a criação dessa plataforma através da conexão dos módulos em `sc_main`. O monitor das propriedades também é declarado na linha 15.

Após a criação dessa plataforma, o código C gerado é adaptado para

Programa 7 Criação dos módulos e conexão para o sistema SCIC.

```
1  #include <iostream>
2  #include <systemc.h>
3  #include "ac_stats_base.H"
4  #include "msp430.H"
5  #include "ac_tlm_mem.h"
6  #include "counter.h"
7
8  #include "monitor.h"
9
10 int sc_main(int ac, char *av[]) {
11
12   //! ISA simulator
13   msp430 msp430_proc1("msp430_1", "random_floats.csv"); // simulador
14
15   monitor mon("mon1", msp430_proc1); // monitores das propriedades
16
17   ac_tlm_mem<2>* mem1; // memória TLM
18   mem1 = new ac_tlm_mem<2>("mem1",128*1024);
19
20   msp430_proc1.DM_port(*(mem1->target_export[0])); // conexão memória+
         processador
21
22   // Periférico para temporização
23   Counter c1("Counter_1", 0);
24   c1.portcpu(msp430_proc1.inta);
25   c1.portDM(*(mem1->target_export[1]));
26
27   ...
28
29   sc_start();
30
31   ...
32
33   return msp430_proc1.ac_exit_status;
34 }
```

que o algoritmo de controle seja executado dentro do período de amostragem especificado para o sistema (10 milissegundos). O modelo do *timer*, criado por Zijlstra (2015), leva em consideração o relógio interno utilizado nos microcontroladores MSP430. Dessa forma, basta habilitar a interrupção por temporização e configurar o valor do registrador de comparação, utilizado para gerar a interrupção a cada período de amostragem. O funcionamento, do ponto de vista do modelo, é bastante simples. Basicamente, toda vez que o *timer* alcança o tempo configurado para uma interrupção, um pedido é gerado através da interface TLM do modelo e um método específico é executado. Esse método anula a instrução sendo executada, salva o *program counter* no topo do *stack* e move a execução para a ISR. No fim dessa interrupção, a instrução `reti` (*return form interrupt*) retorna a execução normal do programa.

Essa descrição simplificada resume as tarefas executadas na Fase 1. É importante observar que o sistema é desenvolvido de forma abstrata, levando em conta um fluxo de projeto de um sistema de controle. Ou seja, nenhuma verificação é realizada no sentido de garantir que o algoritmo de controle

gerado vai ser executado em tempo suficiente (menor 10 milissegundos) no Simulink®. Com a plataforma virtual é possível, sem ter o *hardware* disponível, avaliar essa perspectiva já na Fase 2a1.

5.2.2.2 Procura de Erros de Implementação com o ESBMC

Após simulado, esse sistema é submetido ao ESBMC para procura por erros de implementação (ou de geração do código pelo Simulink®). Como esclarecido na Seção 3.2.3 (Programa 2), se utilizar o ponto de entrada de verificação na função `main()` do código gerado, o ESBMC não analisará o *software* de forma correta, pois o algoritmo de controle é chamado através de uma interrupção (além disso o algoritmo de controle executa em um laço infinito). Tendo isso em mente, é utilizado uma abordagem diferenciada, também para fornecer certa modularidade. Primeiramente, o código gerado é preprocessado pela ferramenta `CIL`[21], utilizando o modelo de máquina mostrado abaixo:

Commando 1 Modelo de máquina utilizado na ferramenta CIL.

```
short=2,2, int=2,2 long=4,2 long_long=8,2 enum=2 void=1 pointer=2,2 float=4,2
double=4,2 long_double=8,2 bool=1,1 fun=1,2 alignof_enum=2 alignof_string=1
max_alignment=2 size_t=int wchar_t=int char_signed=true
const_string_literals=true big_endian=false
__thread_is_keyword=true __builtin_va_list=true underscore_name=false
```

Durante essa fase de preprocessamento, o código é fundido em um arquivo e o modelo de máquina garante que o código será processado levando em consideração as características da arquitetura/compilador do microcontrolador. Esse arquivo, saída da ferramenta CIL, é utilizado como entrada de um *script* em Python que obtém o GSCF como mostrado na Figura 21. Então, o ESBMC é executado em cada uma das funções, a partir das folhas até a raiz, seguindo o fluxograma mostrado na Figura 11 (2b).

Commando 2 Comando utilizado no ESBMC.

```
/usr/bin/time -v -o output/time esbmc ${solver} --overflow-check --memlimit 8g
--timeout 15m --little-endian ${function} ${file} &> resultados/esbmc/file/função
```

Os resultados dessa primeira etapa são resumidos na Tabela 4. Para todos os casos, são consideradas entradas não determinísticas sem limites, em uma tentativa de explorar o máximo possível do sistema através de *Model*

[21] <https://github.com/cil-project/cil>

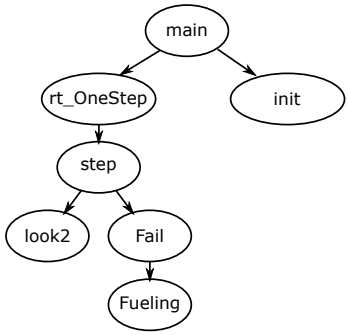

Figura 21 – Grafo Estático de Chamada de Função para o SCIC.

Checking. As funções que não foram verificadas são marcadas para serem posteriormente simuladas.

Tabela 4 – Resultados obtidos das funções Fase 2, procura por erros de implementação: Cenário 1 (C1) 2b; Cenário 2 (S2) novamente etapa 2b considerando informação da Fase 3. CV - Condições de verificação. MO - Memory out. TO - Timeout. "-" desnecessário/não se aplica.

Função			Procura Erros			
Nome	Profundidade	CV(C1/C2)	C1		C2	
			Mem (MB)	Tempo (s)	Mem (MB)	Tempo (s)
rt_OneStep	1	-/249	MO	23,6	193,09	58,98
init	1	57/57	0,42	62,69	0,42	62,69
step	2	-/-	MO	20,84	-[1]	-
look2	3	-/-	MO	21,6	-	-
Fail	3	0/-	63,40	0,19	-	-
Fueling	4	0/-	61,19	0,18	-	-

Nota: As funções `Fail` e `Fueling` implementam as máquinas estados mostradas na Seção 5.1.1.
[1] Com o limite de desenlace o ESBMC é capaz de verificar corretamente considerando o ponto de entrada na função `step`.

Observando os valores durante simulação (Fase 3), foi possível obter o limite máximo de execução dos laços utilizados nas *look up tables* do algoritmo de controle, mostrados no Programa 8 linhas 20, 23, 30 e 33.

Posteriormente, o ESBMC foi executado novamente, considerando os valores de simulação, os resultados obtidos são mostrados como Cenário 2 (C2).

5.2.2.3 Verificação de Propriedades Temporais com ESBMC

Até esse momento, não foram verificadas propriedades temporais do sistema. Somente propriedades de segurança relacionadas a erros típicos e características do programa em C. Com base nisso, foram criadas 11 propri-

Programa 8 Trecho de programa que apresenta laços que causam MO em uma primeira execução do ESBMC.

```
1  real32_T look2_iflf_linlca(real32_T u0, real32_T u1, const real32_T bp0[],
2                             const real32_T bp1[],
3                             const real32_T table[],
4                             const uint32_T maxIndex[],
5                             uint32_T stride) {
6  #ifdef S1
7    u0 = nondet_real32_T();
8    u1 = nondet_real32_T();
9    stride = nondet_uint32_T();
10
11   bp0 = (real32_T const *)malloc(sizeof(real32_T));
12   bp1 = (real32_T const *)malloc(sizeof(real32_T));
13   table = (real32_T const *)malloc(sizeof(real32_T));
14   maxIndex = (uint32_T const *)malloc(sizeof(uint32_T));
15 #endif //ESBMC
16
17   ...
18
19   /* Linear Search */
20   for (bpIdx = maxIndex[0UL] >> 1UL; u0 < bp0[bpIdx]; bpIdx--) {
21   }
22
23   while (u0 >= bp0[bpIdx + 1UL]) {
24     bpIdx++;
25   }
26
27   ...
28
29   /* Linear Search */
30   for (bpIdx = maxIndex[1UL] >> 1UL; u1 < bp1[bpIdx]; bpIdx--) {
31   }
32
33   while (u1 >= bp1[bpIdx + 1UL]) {
34     bpIdx++;
35   }
36
37   ...
38
39   return y;
40 }
```

edades, que cobrem as transições das máquinas de estados e fornecem uma demonstração das capacidades da metodologia sendo aplicada.

As propriedades especificadas são mostradas na Tabela 5. Propriedades com ID de 1 a 4 são especificadas para as cobertura das transições da máquina de estado mostrada na Figura 13; 5 a 8 para cada uma das máquinas na Figura 12; por fim, propriedades 9 a 11 assertam as transições entre os modos de operação do sistema (Figura 14).

Na verificação das propriedade é seguido o fluxo de verificação proposto por Morse (2015). Para cada propriedade é criado um monitor e uma espécie de *testbench* (*test hardness*), como mostrado no Programa 9, que inicia a execução simbólica dos monitores, chamando a função que contém o comportamento a ser verificado `fuel_rate_control_Fail()`. Nesse caso, é considerado a função

que implementa a máquina de estado respectiva às propriedades 1 a 4.

Tabela 5 – Propriedades especificadas para verificação funcional do SCIC.

ID	Propriedades
1	G((fail == none) && (event == inc) ⇒ F(next_state == one))
2	G((fail == one) && (event == inc) ⇒ F(next_state == two))
3	G((fail == 1) && (multi == two) && (event == inc) ⇒ F(next_state == three))
4	G((fail == 1) && (multi == three) && (event == inc) ⇒ F(next_state == four))
5	G((ego > 1.2) ⇒ F(event == inc))
6	G((map > 0.95) || (map < 0.05) ⇒ F(event == inc))
7	G((throttle > 90.0) || (throttle < 3.0) ⇒ F(event == inc))
8	G((speed == 0.0) && (map < 250.0) ⇒ F(event == inc))
9	G((fail == none) ⇒ F(fuel_mode = LOW))
10	G((fail == one) ⇒ F(fuel_mode = RICH))
11	G((fail > one) ⇒ F(fuel_mode = DISABLE))

Programa 9 *Test Harness* criado para verificação das propriedades 1 a 4.

```
1  #include <pthread.h>
2  #include <stdbool.h>
3
4  void fuel_rate_control_initialize(void);
5  void fuel_rate_control_Fail(void);
6  bool nondet_bool();
7
8  pthread_t ltl2ba_start_monitor();
9  void ltl2ba_finish_monitor();
10
11 int main(int argc, char **argv) {
12   pthread_t thread;
13
14   fuel_rate_control_initialize();
15
16   thread = ltl2ba_start_monitor();
17
18   while (1) {
19     fuel_rate_control_Fail();
20   }
21
22   ltl2ba_finish_monitor(thread);
23
24   return 0;
25 }
```

Commando 3 Comando utilizado para execução do ESBMC na etapa de verificação funcional.

```
/usr/bin/time -v -o results/time_stats_ID esbmc main_test.c mon_ID.c
../../fuel_rate_control.c --ltl --unwind X --partial-loops --z3
--no-pointer-check --no-bounds-check --no-unwinding-assertions
--no-div-by-zero-check --memlimit 8g --timeout 15m --unwindset 13,Y &> resultados
```

Como sugerido por Morse (2015), os valores de `X` e `Y` (ver **??** 3) devem ser incrementados de acordo, pois a validade da propriedade pode apresentar dependência do limite de desenlace do programa. Foi observado tal comportamento para as propriedades 1 a 4. Os resultados mostrados na Tabela 6 representam a execução do ESBMC com valores `X=2` e `Y=10`. A ideia é demonstrar situações em que devido a complexidade, propriedades que assertam o comportamento a nível de sistema, através da função `step()` (ver Figura 21), acabam não sendo verificadas pelo ESBMC. Evidentemente que em fases posteriores de projeto análises quantitativas e conclusivas devem ser conduzidas para validar o comportamento do sistema.

A partir desses resultados são realizadas simulações e monitoramento das propriedades que não foram verificadas pelo ESBMC (5 a 11), na Fase 3 da metodologia, como descrito na próxima seção.

Tabela 6 – Verificação das propriedades (Tabela 5) com ESBMC. MO - *Memory out*. TO - *Timeout*. (*) - Monitoradas Posteriormente Através de Simulações.

ID	ESBMC	
	Mem (MB)	Tempo (s)
1	170,08	212
2	169,06	268
3	179,24	308
4	177,48	376
5*	2764,99	TO
6*	2760,01	TO
7*	2758,90	TO
8*	2816,28	TO
9*	MO	18,35
10*	MO	16,89
11*	MO	16,79

5.2.2.4 Simulação e Monitoramento das Propriedades

Dois cenários distintos de simulação são considerados para esse sistema. O primeiro considera os sensores operando normalmente, ou seja, deve-se operar sem entrar em estado de falha. No segundo cenário, são inseridas falhas propositais nos sensores (através dos estímulos de entrada) de forma a avaliar se as propriedades se mantém. Em ambos os casos são obtidos 100000

valores randômicos[22] (distribuição uniforme) através de um programa simples, implementado utilizando a biblioteca `Boost`[23]. Os valores são criados de forma estática, antes de realizar a simulação, e armazenados em formato de texto.

A interface com o *software* embarcado é realizada através do modelo; sincronização é realizada levando em conta o contador de programa de entrada na interrupção e o endereço de retorno. Na entrada são inseridos os estímulos diretamente na memória simulada. Quando a função retorna, as propriedades são amostradas, pois o algoritmo de controle foi executado.

Os resultados da execução da plataforma completa são mostrados na Tabela 7. Em todos os casos foram introduzidas erros propositais no código de forma a confirmar se as propriedades estavam sendo avaliadas corretamente. Essa mesma ideia também foi aplicada quando verificando com o ESBMC.

Tabela 7 – Resultados obtidos durante simulação do SCIC, considerando monitores para propriedades 5 a 11.

	ΣInstruções	Tempo (s)	Memória (MB)	MIPS
Sem Falhas	3641175703	88,49	5,46	41,33
Com Falhas	2596564644	65,17	5,64	40,03

Durante as simulações não foram observados erros, exceto quando inseridos propositalmente. Os valores na Tabela 7 mostram que a plataforma executa em tempo consideravelmente rápido, mantendo certo *overhead* dos monitores.

5.2.3 Plataforma Virtual para o SCAE

Os resultados mostrados anteriormente foram obtidos considerando simulações com valores aleatórios como estímulo de entrada para o *software*. Há casos, no entanto, em que é necessário utilizar esquemas mais elaborados para teste do sistema, com entradas que possuam significado dentro do contexto funcional sendo desenvolvido.

O SCAE fornece um estudo de caso ideal para demonstrar outras formas de estimular o *software* embarcado, sem introduzir grande *overhead* de implementação. Seguindo o mesmo princípio mostrado anteriormente é gerado o código para o controlador, através do Simulink®. Após essa etapa, a plataforma virtual mostrada na Figura 22 é montada para simulação, considerando os mesmos executáveis que seriam utilizados no *hardware*. Certas propriedades

[22]

[23] <http://www.boost.org/>

de interesse podem ser inspecionadas, ainda sem considerar verificação através do ESBMC ou monitoramento de propriedades temporais, como: habilidade do sistema em executar dentro do período de amostragem considerado (1 milissegundo); precisão dos valores calculados; possíveis diferenças entre as simulações podem ser depuradas.

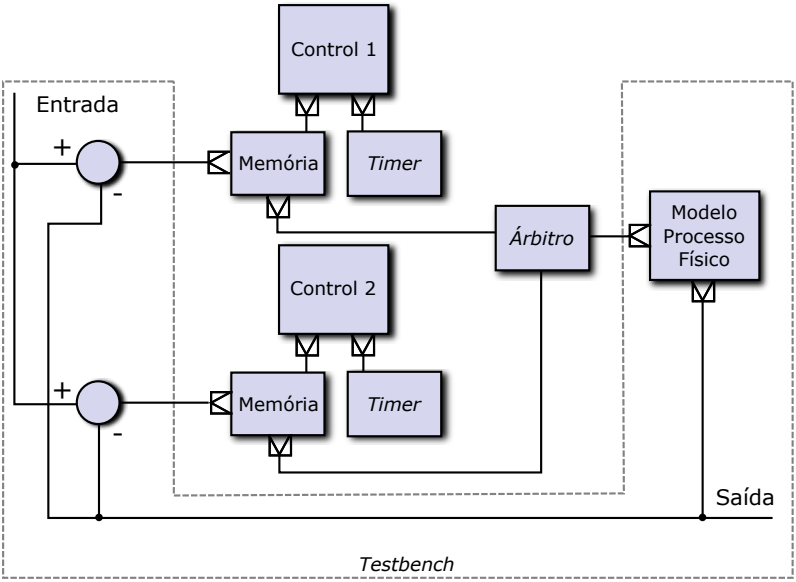

Figura 22 – *Testbench* montado para simulação do SCAE. Notar a complexidade maior por parte do *testbench* que inclui também o modelo do processo físico.

A Figura 23 mostra um comparação entre as duas simulações. MBD realizada no Simulink® e ESL considerando a plataforma virtual.

Deve-se notar a natureza das simulações, destacando as diferentes características das metodologias de desenvolvimento, como comentado anteriormente. Além disso, nota-se certa diferença entre os resultados mostrados para as duas simulações, como pode ser visto no gráfico que mostra a diferença absoluta (ver Figura 24). No caso da simulação ESL, é evidente que o *software* embarcado é responsável pelo cálculo do sinal de controle, ou seja, é utilizado uma representação considerando os 16 bits da arquitetura do microcontrolador. Apesar disso, o padrão de representação utilizado para valores `float` não introduz erros tão consideráveis, em comparação com os resultados apresentados por MBD. Essa diferença pode ter ocorrido devido ao esquemas de temporização utilizados, através de eventos.

Esses resultados mostram as diferenças essenciais entre as metodologias.

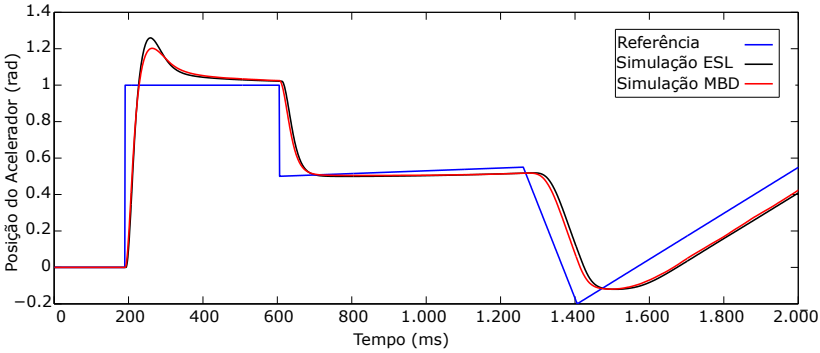

Figura 23 – Comparação entre as duas simulações realizadas em MBD e ESL (plataforma virtual) para as mesmas entradas.

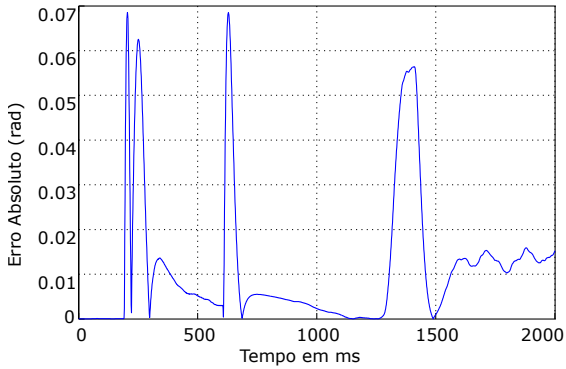

Figura 24 – Diferença absoluta entre os sinais de controle obtidos com MBD e ESL.

Por um lado MBD fornece um ambiente mais robusto para projeto de sistemas baseados em algoritmos matemáticos. Porém, para simulações a nível de sistema, considerando PIL ou SIL, ESL possui vantagens, devido as facilidades, em comparação com MBD, em modelagem e simulação de *hardware* e *software*.

5.2.4 Simulação e Verificação do Computador de Bordo do *CubeSat*

As especificações apresentadas na Seção 5.1.3.2 definem as funcionalidades pretendidas para o computador de bordo. Com base nessas especificação foram criadas aplicações, utilizando a plataforma descrita (ver Figura 16) para depuração e simulação.

As tarefas realizadas pelo OBDH são essencialmente temporais. Ou seja, o mesmo modelo utilizado para fornecer interrupções por tempo, nos

estudos de caso do sistemas de controle (SCIC e SCAE), é reutilizado para fornecer o *tick* que governa a execução do sistema operacional. Essa situação gera um problema, pois a execução é determinada por uma interrupção temporal, a qual, deve ser levada em conta para verificação do sistema. Além disso, o sistema completo (sistema operacional e aplicação), possui complexidade considerável, cerca de 3800 linhas de código. Com base nessas informações, optou-se por focar na Fase 3 da metodologia proposta, para esse caso específico, e demonstrar perspetivas de verificação considerando simulações com ferramentas comerciais.

Levando em conta a especificação do sistema, foi criado um *testbench* que imita o comportamento da ETR, no que diz respeito a visibilidade de comunicação. As propriedades foram especificadas utilizando as ferramentas comerciais da empresa Cadence®, na linguagem Property Specification Language (PSL). Essa linguagem de especificação também é baseada em um modelo linear de tempo, assim como LTL. Sua aplicação, no entanto, é muito focada na asserção de propriedades típicas de *hardware*. Os operadores temporais básicos são os mesmos, exceto por alguns operadores especializados para *hardware*. Por ser uma ferramenta comercial há suporte a grande número de asserções possíveis (e.g. SEREs - *Sequential Extended Regular Expressions*). No entanto, devido as restrições do ponto de vista de *software*, é necessário criação de eventos e sincronização cuidadosa entre *testbench* e *software* embarcado. O que acaba limitando a utilização de certos operadores, como *Next*. Mesmo considerando o mesmo formato de propriedade utilizado anteriormente (i.e, pergunta-resposta), é possível observar propriedades cruciais do sistema, além de capturar o erro inserido na especificação de forma satisfatória.

A propriedade mais crítica do sistema implementado, pode ser traduzida em uma propriedade de segurança, que asserta que o TTC nunca deve estar desligado quando a ETR estiver visível. Além disso, propriedades interessantes podem ser extraídas do protocolo de comunicação. Considerando que as variáveis `obdh_tx`, `obdh_rx` e `ttc_pwr_state` representam respectivamente: *buffer* de transmissão I2C do OBDH; *buffer* de recepção I2C do OBDH; *status* do TTC; a tabela Tabela 8 resume as propriedades especificadas para o sistema. Propriedades 1 a 4 assertam o comportamento esperado do protocolo de comunicação utilizado. Notar que as propriedades 3 e 4 dizem respeito a variáveis do multisistema OBDH e do TTC, ou seja, *software* sendo executados por processadores/simuladores diferentes.

Seguindo o fluxo de verificação das ferramentas sendo utilizadas para este caso, é necessário associar um evento para cada propriedade a ser amostrada. Muito similar ao que já havia sido realizado com os monitores em LTL. Para propriedades 1 a 4 é utilizado as interrupções do I2C como origem dos

Tabela 8 – Propriedades especificadas para verificação funcional do computador de bordo.

ID	Propriedades
1	G((obdh_tx == ask_ground_status) ⇒ F((obdh_rx == INVISIVEL) || (obdh_rx == VISIVEL)))
2	G((obdh_tx == ask_pwr_status) ⇒ F((obdh_rx == ON) || (obdh_rx == OFF)))
3	G((obdh_tx == set_ttc_on) ⇒ F(ttc_pwr_state == ON))
4	G((obdh_tx == set_ttc_off) ⇒ F(ttc_pwr_state == OFF))
5	G(ttc_pwr_status == ON)

eventos, e para propriedade 5 o comportamento simulado da ETR gera um evento para amostragem. Os resultados obtidos da avaliação dessas propriedades são mostrados de forma "crua" no Programa 10. Foram simulados 10 períodos orbitais, considerando uma janela de visibilidade de 8 milissegundos.[24] A propriedade 5 falha em 9 dos 10 períodos, justamente após o sistema ter inicializado (alcançado estado S2 na Figura 17). Para as outras propriedades não é observado nenhuma falha.

Programa 10 Resultados obtidos da simulação da plataforma virtual do *CubeSat*.

```
SC simulation stopped by user.

SystemC : SystemC stopped at time 580
ncsim> assertion -summary
  Disabled Finish Failed   Assertion Name
         0    127      0   sc_main.testbench_1.ground_status_req_rps
         0      0      0   sc_main.testbench_1.pwr_status_req_rps
         0     56      0   sc_main.testbench_1.pwr_status_set_off
         0     21      0   sc_main.testbench_1.pwr_status_set_on
         0      1      9   sc_main.testbench_1.
     safety_check_ttc_status
  Total Assertions = 5,  Failing Assertions = 1,  Unchecked
     Assertions = 1
  Assertion summary at time 580 MS + 0
ncsim> exit
```

O desempenho de simulação é consideravelmente comprometido quando utilizando plataformas tão complexas. Nesse caso a simulação toda consumiu cerca de 315,7 segundos no total (cerca de 5,3 minutos) e alcançou 8,07 MIPS. Uma diferença considerável em comparação com as outras plataformas.

Uma questão interessante é que a propriedade denominada: `sc_main.testbench_1.pwr_status_req_rps` (na realidade a propriedade com ID 2 na Tabela 8) não foi avaliada nenhuma vez. Esse resultado revela uma perspectiva interessante de análise, pois essa propriedade não foi avaliada porque nenhum pedido do tipo `ask_pwr_status` foi realizado pelo OBDH. A

[24] O que corresponderia a aproximadamente 13 minutos no tempo real.

forma que a aplicação foi implementada permite deduzir o estado do TTC implicitamente. Esse resultado revela a flexibilidade das simulações em revelar caraterísticas importantes do sistema, considerando também outras perspectivas além da funcional.

5.3 Conclusões e Discussão

Neste capítulo foram apresentados os estudos de caso e relacionados os resultados obtidos. Como a metodologia proposta para desenvolvimento e verificação pode ser aplicada de diversas formas, explorando cenários diferentes para cada um dos sistemas, foram demonstrados diversas possibilidades de aplicação. Os resultados foram agrupados de forma a fornecer uma visão clara e concisa, abordando pontos específicos e gerais.

Algumas considerações finais quanto a cada um dos estudos de casos. Primeiramente, o modelo do microcontrolador foi desenvolvido em fases iniciais de implementações deste trabalho e embora tenha sido verificado e executado um número considerável de instruções, ainda possui algumas desconformidades. Durante a modelagem do MSP430, o grande número de combinações de cada uma das instruções, nos modos de endereçamento, fez com que fosse utilizado um esquema não tradicional de modelagem com ArchC. Muitas instruções são decodificadas e, posteriormente, é determinado o seu comportamento. O ideal seria que cada instrução tivesse seu comportamento definido pela decodificação, sem *overhead* durante a execução. As mudanças devem ser realizadas na descrição do modelo, o que as tornará consideravelmente maiores. Por outro lado, ganho significativo em desempenho pode ser alcançado realizando tais adequações.

Com relação a Fase 2 da metodologia e aplicação do ESBMC ao sistema SCIC. Durante os testes foram realizados diversas experimentações, utilizando todos os *solvers*, inclusive com outro *model checker* (`CBMC`[25]). Observou-se que, em geral, o ESBMC gera maior número de condições de verificação e ainda oferece a possibilidade de verificação temporal. Não era objetivo inicial deste trabalho realizar comparações ou análises quantitativas, somente expor a metodologia e estressá-la a diferentes contextos de utilização.

É importante deixar claro as contribuições com relação a combinação de abordagens de forma unificada, levando em conta tendências recentes e o estado da arte em verificação e desenvolvimento de *software* embarcado.

[25] <http://www.cprover.org/cbmc/>

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, foi apresentado uma metodologia para verificação de *software* embarcado considerando fases iniciais de projeto. Essa metodologia foi dividida em três fases principais e faz uso de métodos híbridos de verificação. Perante ao estado da arte, foi utilizado ferramentas e abordagens recentes nas implementações realizadas.

Como resultados, foi aplicado a metodologia de verificação a um sistema de controle de injeção de combustível, gerado automaticamente através de abordagens *Model Based Design*. Os resultados obtidos mostram que é possível aplicar a metodologia já em fases iniciais, sem considerar sistemas completos, e utilizar simulação e *Model Checking* de forma combinada, mesmo antes de se ter o *hardware* final. Diferentes cenários podem ser explorados dependendo do contexto em que se deseja aplicar a metodologia. Foram apresentados dois casos específicos que estimulam fases diferentes da metodologia proposta. Um esquema de simulação do sistema que explora as diferenças entre simulação em ESL e MBD, com o sistema de controle de acelerador eletrônico. Além disso, foram relacionados resultados de monitoramento de propriedades de um sistema consideravelmente complexo, contendo: várias unidades de processamento; barramento de comunicação I2C; protocolo de comunicação; além da utilização de um sistema operacional de tempo real. Esse estudo de caso revela o potencial das simulações a nível de sistema, tanto para monitoramento de propriedades quanto para obtenção de informações valiosas quanto ao comportamento do sistema.

Em geral a combinação de *Electronic System Level Design*, *Model Based Design*, *Model Checking* e verificação de propriedades temporais tem potencial significativo para evidenciar possíveis de erros de implementação, com possibilidade de iniciar desenvolvimento e verificação incremental de forma antecipada.

6.1 Contribuições

As principais contribuições dessa dissertação estão relacionadas a metodologia proposta, as aplicações aos estudos de caso e todo desenvolvimento necessário para alcançar tais metas.

Vale ressaltar, do ponto de vista teórico, a identificação de uma lacuna de trabalhos que abordem métodos híbridos de verificação, considerando desenvolvimento de simuladores do conjunto de instruções como parte de suas metodologias. Dessa forma, os aspectos menos explorados no estado da arte foram incorporados na metodologia proposta, sendo assim, a principal

contribuição teórica deste trabalho.

Como contribuições técnicas pode-se destacar principalmente: desenvolvimento e verificação do simulador para o microcontrolador MSP430, utilizado em outro trabalho de mestrado (ver Zijlstra (2015)); aplicação das ferramentas nos estudos de caso; criação da plataforma virtual para o sistema de controle de posição de acelerador eletrônico; implementação e verificação do computador de bordo do *CubeSat*.

6.2 Trabalhos Futuros

Como trabalhos futuros deve ser considerado melhorias na modelagem por parte do simulador do conjunto de instruções, principalmente para permitir simulações compiladas e melhoras em desempenho. Por parte da metodologia de verificação, embora soluções comerciais, como mostrado no estudo de caso do computador de bordo, forneçam suporte para verificação de propriedades complexas, futuras implementações podem considerar monitoramento de propriedades mais complexas sem necessidade das ferramentas comerciais. Para isso, pode ser necessário modificações na implementação interna do SystemC.

Com o desenvolvimento do *software* do projeto FloripaSat pode-se aplicar essa metodologia para verificação do sistema todo, justamente como mostrado no estudo de caso, porém considerando a versão de vôo do sistema.

Futuras implementações também podem ser realizadas para automação de diversos processos realizados manualmente, até mesmo considerando integração entre simulação e *Model Checking* de forma automática.

REFERÊNCIAS

Aalborg University's Student Satellite. *Design of Hardware and Software for the Powersupply for AAU CubeSat*. [S.l.], Novembro 2002. Disponível em: <http://www.space.aau.dk/cubesat/documents/psu/new{_}psu.pdf>.

AARNO, D.; ENGBLOM, J. *Software and System Development using Virtual Platforms: Full-System Simulation with Wind River Simics*. Elsevier Science, 2014. 1–19 p. ISBN 9780128008133. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/B9780128007259000019>.

ACCELLERA. *About SystemC: The Language for System-Level Modeling, Design and Verification*. Out. 2015. Acessado em: 22 de Out. 2015. Disponível em: <http://accellera.org/community/systemc/about-systemc>.

ALEXANDER, P. *System Level Design with Rosetta*. Elsevier Science, 2011. (Systems on Silicon). ISBN 9780080498379. Disponível em: <https://books.google.com.br/books?id=z_o0X6bDhJwC>.

AZEVEDO, R.; ALBERTINI, B.; RIGO, S. ARP: Um Gerenciador de Pacotes para Sistemas Embarcados com Processadores Modelados em ArchC. p. 69–71, 2010.

BAIER, C.; KATOEN, J.-P. *Principles Of Model Checking*. MIT Press, 2008. I–XVII, 1–5 p. ISSN 00155713. ISBN 9780262026499. Disponível em: <http://mitpress.mit.edu/books/principles-model-checking>.

BARNASCONI, M. SystemC AMS Extensions : Solving the Need for Speed. *47 Design Automation Conference (DAC)*, p. 1–8, 2010.

BARTHOLOMEU, M. *Simulação Compilada para Arquiteturas Descritas em ArchC*. Tese (Doutorado) — UNICAMP, Programa de Pós-Graduação em Ciência da Computação, Nov 2005.

BECKER, M.; KUZNIK, C.; MUELLER, W. Virtual Platforms for Model-Based Design of Dependable Cyber-Physical System Software. In: *2014 17th Euromicro Conference on Digital System Design*. IEEE, 2014. p. 246–253. ISBN 978-1-4799-5793-4. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6927251>.

BEHREND, J. et al. Scalable and Optimized Hybrid Verification of Embedded Software. *Journal of Electronic Testing*, v. 31, n. 2, p. 151–166, 2015. ISSN 0923-8174. Disponível em: <http://link.springer.com/10.1007/s10836-015-5518-4>.

BIERE, A. Bounded model checking. In: Armin Biere, Marijn Heule, Hans van Maaren, T. W. (Ed.). *Handbook of Satisfiability*. IOS Press, 2009. v. 185, n. 1, cap. 14, p. 457–481. ISBN 9781586039295. Disponível em: <http://ebooks.iospress.nl/volume/handbook-of-satisfiability>.

CADAR, C.; DUNBAR, D.; ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 209–224. Disponível em: <http://dl.acm.org/citation.cfm?id=1855741.1855756>.

Cadence Design Systems. *Incisive Enterprise Specman Products*. Mar. 2016. Acessado em: 14 de Mar. 2016. Disponível em: <https://www.cadence.com/rl/Resources/datasheets/specman_elite_ds.pdf>.

CARDOSO, R. A. *Desafios no desenvolvimento de plataformas capazes de executar sistemas operacionais utilizando o ArchC*. Dissertação (Mestrado) — UNICAMP, Programa de Pós-Graduação em Ciência da Computação, Fev. 2015.

CHRISTIAN, K. *Enhancing Embedded Systems Simulation: A Chip-Hardware-in-the-Loop Simulation Framework*. Dissertação (Mestrado) — Technische Universität M¨unchen, 2010. Disponível em: <http://www.springerlink.com/index/10.1007/978-3-8348-9916-3>.

CISCO. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update The Cisco ® Visual Networking Index (VNI) Global Mobile Data Traffic Forecast Update*. [S.l.], 2015. Disponível em: <http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.pdf>.

CLARKE, E. et al. Model checking and the state explosion problem. In: MEYER, B.; NORDIO, M. (Ed.). *Tools for Practical Software Verification*. Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7682). p. 1–30. ISBN 978-3-642-35745-9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-35746-6_1>.

CLARKE, E. M. The Birth of Model Checking. In: *25 Years of Model Checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. v. 5000 LNCS, p. 1–26. ISBN 3540698493. Disponível em: <http://link.springer.com/10.1007/978-3-540-69850-0{_}1>.

CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In: *Logics of Programs*. Springer-Verlag, 1981. p. 52–71. ISBN 3-540-11212-X. Disponível em: <http://www.springerlink.com/index/10.1007/BFb0025774>.

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1983. (POPL '83), p. 117–126. ISBN 0-89791-090-7. Disponível em: <http://doi.acm.org/10.1145/567067.567080>.

CONRAD, M.; MOSTERMAN, P. J. Model-based design using simulink – modeling, code generation, verification, and validation. In: ____. *Formal*

Methods. John Wiley & Sons, Inc., 2013. p. 159–181. ISBN 9781118561898. Disponível em: <http://dx.doi.org/10.1002/9781118561898.ch4>.

CORDEIRO, L. et al. Semiformal Verification of Embedded Software in Medical Devices Considering Stringent Hardware Constraints. *2009 International Conference on Embedded Software and Systems*, p. 396–403, 2009. Disponível em: <http://eprints.ecs.soton.ac.uk/17239/>.

CORDEIRO, L. C. *SMT-Based Bounded Model Checking of Multi-threaded Software in Embedded Systems*. Tese (Doutorado) — Universidade de Southampton, Southampton, Hampshire, Inglaterra, Abr. 2011.

COUNCIL, N. R.; JACKSON, D.; THOMAS, M. *Software for Dependable Systems: Sufficient Evidence?* Washington, DC, USA: National Academy Press, 2007. ISBN 0309103940, 9780309103947.

DELAMARO, M.; MALDONADO, J.; JINO, M. *Introdução ao teste de software*. Campus, RJ, Brasil, 2007. ISBN 9788535226348. Disponível em: <https://books.google.com.br/books?id=7R6XPgAACAAJ>.

D'SILVA, V.; KROENING, D.; WEISSENBACHER, G. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 27, n. 7, p. 1165–1178, July 2008. ISSN 0278-0070.

DUTTA, S.; TABAKOV, D.; VARDI, M. Y. CHIMP: A tool for assertion-based dynamic verification of systemc models. *CEUR Workshop Proceedings*, v. 1130, p. 38–45, 2014. Disponível em: <http://www.scopus.com/inward/record.url?eid=2-s2.0-84908307368{&}partnerID=40{&}md5=38963e1e2802441257318fa7077f0ff7>.

ECKER, W. et al. Implementation of a Transaction Level Assertion Framework in SystemC. In: *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007. p. 1–6. ISBN 978-3-9810801-2-4. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4211916>.

FDA. *Medical Device Recall Report*. [S.l.], 2012. Disponível em: <http://www.fda.gov/downloads/aboutfda/centersoffices/officeofmedicalproductsandtobacco/cdrh/cdrhtransparency/ucm388442.pdf>.

FISHER, M. *An Introduction to Practical Formal Methods Using Temporal Logic*. Wiley, 2011. ISBN 9781119991465. Disponível em: <https://books.google.com.br/books?id=zl6OLZv7d1kC>.

GEORGAKOS, G.; SCHLICHTMANN, U.; SCHNEIDER, R. Reliability challenges for electric vehicles: From devices to architecture and systems software. In: *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*. [S.l.: s.n.], 2013. p. 1–9. ISSN 0738-100X.

GODEFROID, P.; KLARLUND, N.; SEN, K. DART. *ACM SIGPLAN Notices*, v. 40, n. 6, p. 213, jun 2005. ISSN 03621340. Disponível em: <http://dl.acm.org/citation.cfm?id=1064978.1065036http://portal.acm.org/citation.cfm?doid=1064978.1065036>.

HAGAR, J. D. et al. Introducing Combinatorial Testing in a Large Organization. *Computer*, v. 48, n. 4, p. 64–72, apr 2015. ISSN 0018-9162. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7085645>.

HARRISON, J. *Handbook of Practical Logic and Automated Reasoning*. 1st. ed. New York, NY, USA: Cambridge University Press, 2009. ISBN 0521899575, 9780521899574.

IBM. *IBM Flowcharting Techniques*. 1970. Acessado em: 08 de Fev. 2016. Disponível em: <http://www.eah-jena.de/~kleine/history/software/IBM-FlowchartingTechniques-GC20-8152-1.pdf>.

IEEE. *Rosetta WG - Rosetta Systems Level Design Language Working Group*. [S.l.], 2008. Em desenvolvimento. Disponível em: <http://standards.ieee.org/develop/wg/Rosetta_WG.html>.

IEEE. *1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual*. 2011. ed. [S.l.], sep 2011. Disponível em: <https://standards.ieee.org/findstds/standard/1666-2011.html>.

INSTRUMENTS, T. *MSP Low-Power Microcontrollers*. [S.l.], Janeiro 2016. Disponível em: <http://www.ti.com/lit/sg/slab034ac/slab034ac.pdf>.

ISERMANN, R.; SCHWARZ, R.; STÖLZL, S. Fault-tolerant drive-by-wire systems. *IEEE Control Systems Magazine*, v. 22, n. 5, p. 64–81, 2002. ISSN 02721708.

JACKSON, M. Seeing more of the world [requirements engineering]. *Software, IEEE*, v. 21, n. 6, p. 83–85, Nov 2004. ISSN 0740-7459.

KROENING, D. et al. Effective verification of low-level software with nested interrupts. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. San Jose, CA, USA: EDA Consortium, 2015. (DATE '15), p. 229–234. ISBN 978-3-9815370-4-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2755753.2755803>.

LETTNIN, D. Verification of temporal properties in embedded software. 2009.

LETTNIN, D. et al. Coverage driven verification applied to embedded software. *Proceedings - IEEE Computer Society Annual Symposium on VLSI: Emerging VLSI Technologies and Architectures*, p. 159–164, 2007.

LIN, Y. et al. A Divergence-Oriented Approach to Adaptive Random Testing of Java Programs. In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009. p. 221–232. ISBN 978-1-4244-5259-0. ISSN 1938-4300. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5431768>.

MARTIN, G.; BAILEY, B.; PIZIALI, A. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Elsevier Science, 2010. (Systems on Silicon). ISBN 9780080488837. Disponível em: <https://books.google.com.br/books?id=InTpLMpMMHAC>.

MATHWORKS. *Simulink Design Verifier*. Out. 2015. Acessado em: 21 de Out. 2015. Disponível em: <http://www.mathworks.com/products/sldesignverifier/>.

MathWorks. *Simulink Design Verifier*. Mar. 2016. Acessado em: 14 de Mar. 2016. Disponível em: <http://www.mathworks.com/products/simulink/>.

MEENAKSHI, B.; BHATNAGAR, A.; ROY, S. Tool for Translating Simulink Models into Input Language of a Model Checker. In: *Formal Methods and Software . . . .* Heidelberg: Springer-Verlag Berlin, 2006. p. 606–620. ISBN 3540474609. Disponível em: <http://link.springer.com/chapter/10.1007/11901433{_}33>.

MONTOREANO, M. *Transaction Level Modeling using OSCI TLM 2.0*. [S.l.], Maio 2007. 5 p.

MORSE, J. *Expressive and efficient bounded model checking of concurrent software*. Tese (Doutorado) — Universidade de Southampton, Southampton, Hampshire, Inglaterra, Abr. 2015.

MORSE, J. et al. Model checking LTL properties over ANSI-C programs with bounded traces. *Software and Systems Modeling*, p. 1–17, 2013. ISSN 16191366.

MYERS, G. et al. *The Art of Software Testing*. Wiley, 2004. (Business Data Processing: A Wiley Series). ISBN 9780471678359. Disponível em: <https://books.google.com.br/books?id=86rz6UExDEEC>.

NASA. *Escape to Failure: The Qantas Flight 32 Uncontained Engine Failure*. [S.l.], 2015. Disponível em: <https://nsc.nasa.gov/SFCS/SystemFailureCaseStudyFile/Download/579>.

NIE, C.; LEUNG, H. A survey of combinatorial testing. *ACM Computing Surveys*, v. 43, n. 2, p. 1–29, jan 2011. ISSN 03600300. Disponível em: <http://portal.acm.org/citation.cfm?doid=1883612.1883618>.

ORSO, A.; ROTHERMEL, G. Software testing: a research travelogue (2000–2014). *Proceedings of the on Future of Software Engineering - FOSE 2014*, p. 117–132, 2014. Disponível em: <http://dl.acm.org/citation.cfm?id=2593882.2593885>.

OSCI. *SystemC AMS Day 2011 Industry Adoption of the SystemC AMS Standard*. [S.l.], 2011. 65 p.

PAIVA, A. et al. Successful Use of Incremental BMC in the Automotive Industry. *Formal Methods for Industrial Critical Systems*, v. 4916, n. 295311, p. 218 – 233, 2008. Disponível em: <http://www.springerlink.com/index/10.1007/978-3-540-79707-4>.

PIERRE, L.; FERRO, L. A Tractable and Fast Method for Monitoring SystemC TLM Specifications. *IEEE Transactions on Computers*, v. 57, n. 10, p. 1346–1356, 2008. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4509420>.

PIERRE, L. et al. Integrating PSL properties into SystemC transactional modeling - Application to the verification of a modem SoC. In: *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*. [S.l.: s.n.], 2012. p. 220–228. ISBN VO -.

QUEILLE, J. P.; SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In: *International Symposium on Programming SE - 22*. [s.n.], 1982. v. 137, p. 337–351. ISBN 3540114947. Disponível em: <http://link.springer.com/chapter/10.1007/3-540-11494-7{_}22http://link.springer.com/10.1007/3-540-11494-7{_}22>.

RIGO, S.; AZEVEDO, R.; SANTOS, L. *Electronic System Level Design: An Open-Source Approach*. Springer Netherlands, 2011. ISBN 9781402099403. Disponível em: <https://books.google.com.br/books?id=vl3thliIK1YC>.

ROCHA, H. O. *Verificação de Sistemas de Software baseada em Transformações de Código usando Bounded Model Checking*. Tese (Doutorado) — Instituto de Computação - ICOMP, Universidade Federal do Amazonas (UFAM), Jul. 2015.

SEN, K.; MARINOV, D.; AGHA, G. CUTE: A Concolic Unit Testing Engine for C. *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, p. 263, 2005. ISSN 01635948. Disponível em: <http://portal.acm.org/citation.cfm?doid=1081706.1081750>.

SILVA, A. da et al. Runtime Instrumentation of SystemC/TLM2 Interfaces for Fault Tolerance Requirements Verification in Software Cosimulation. *Modelling and Simulation in Engineering*, v. 2014, p. 1–15, 2014. ISSN 1687-5591. Disponível em: <http://www.hindawi.com/journals/mse/2014/105051/>.

SMITH, J.; NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 1558609105.

SPIN. *Verifying Multi-threaded Software with Spin*. Out. 2015. Acessado em: 27 de Out. 2015. Disponível em: <http://spinroot.com/spin/whatispin.html>.

STAATS, M.; HEIMDAHL, M. P. E. Partial Translation Verification for Untrusted Code-Generators. In: *Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 226–237. Disponível em: <http://link.springer.com/10.1007/978-3-540-88194-0{_}15>.

TABAKOV, D.; ROZIER, K. Y.; VARDI, M. Y. Optimized temporal monitors for SystemC. *Formal Methods in System Design*, v. 41, p. 236–268, 2012. ISSN 09259856.

TAPPENDEN, A. F.; MILLER, J. A Novel Evolutionary Approach for Adaptive Random Testing. *IEEE Transactions on Reliability*, John Wiley & Sons, Inc., Hoboken, NJ, USA, v. 58, n. 4, p. 619–633, dec 2009. ISSN 0018-9529. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.77.5425{&}rep=rep1{&}type=pdfhttp://doi.wiley.com/10.1002/0471028959.sof268http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5338642>.

UBM Tech. 2015 Embedded Markets Study: Changes in Today's Design, Development & Processing Environments. Estudo conduzido pela empresa UBM Tech. 2015. Disponível em: <http://webpages.uncc.edu/~jmconrad/ECGR4101-2015-08/Notes/UBM%20Tech%202015%20Presentation%20of%20Embedded%20Markets%20Study%20World%20Day1.pdf>.

VILLA, P. R. C. et al. A complete cubesat mission: the floripa-sat experience. In: *1st Latin American IAA CubeSat Workshop*. [S.l.: s.n.], 2014.

WIELS, V. et al. Formal verification of critical aerospace software. *AerospaceLab*, 2012.

Zhaorong Xiong; Jinian Bian; Yanni Zhao. An assertion-based verification method for SystemC TLM. In: *2010 International Conference on Communications, Circuits and Systems (ICCCAS)*. IEEE, 2010. p. 842–846. ISBN 978-1-4244-8224-5. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5581859>.

ZIJLSTRA, D. *Virtual Satellite Platform of an On-board Computer for Space Applications*. Dissertação (Mestrado) — UFSC, Programa de Pós-Graduação em Engenharia Elétrica, Jun. 2015.

APÊNDICE A – MODELAGEM DE SISTEMAS COM SYSTEMC

A.1 Visão Geral de SystemC

Uma forma bastante simples de visualizar estruturalmente SystemC é utilizar uma representação abstrata em forma de camadas. Esse tipo de representação é muito utilizado em diversas áreas, e foi adaptado nesse trabalho para melhor corresponder ao contexto aqui abordado. Assim, a arquitetura pode ser vista de forma esquemática como mostrado na Figura 25. SystemC é construído sobre C++ e como consequência permite a utilização das camadas mais inferiores no diagrama sem necessidade de utilizar as definições propostas pelo padrão SystemC. Isso faz com que SystemC seja uma solução muito robusta para ser utilizada por desenvolvedores já familiarizados com C++, além é claro de fornecer uma estrutura para modificações e adição de novos elementos. É o caso das extensões propostas *Transaction Level Modeling* (TLM), *Analog/Mixed-Signal* (AMS), *SystemC Verification* (SCV) e *Configuration, Control and Inspection* (CCI).

Figura 25 – Visão Estrutural de SystemC. Adaptado de (ACCELLERA, 2015).

SystemC como um todo pode ser dividido em grupos de elementos funcionais, os quais são responsáveis por tarefas distintas durante a especificação e simulação dos modelos. Na Figura 25 é feita a distinção clara dos

elementos que formam a núcleo de SystemC. Uma descrição breve de suas funcionalidades é apresentada na sequência.

- **Elementos Estruturais:** Os elementos estruturais da são responsáveis pela criação da hierarquia de módulos, bem como conexões para comunicação ou interface;
- **Canais Predefinidos:** São elementos para gerenciamento de recursos e também sincronização entre processos concorrentes, tipicamente utilizados em SystemC;
- **Utilidades:** Elementos utilizados para depuração e criação de formas de onda em formatos conhecidos como *Value Change Dump* (VCD), utilizados normalmente em modelagem RTL;
- **Tipos de dados:** São tipos predefinidos para lidar com valores possíveis em hardware, por exemplo, como alta impedância (Z) ou valores *three-state logic*.

Além desses elementos estruturais é distribuído um simulador embutido, popularmente conhecido como *OSCI simulator*.

Ao topo são criadas as bibliotecas de metodologias, que representam um conjunto de extensões construídas sobre o núcleo de SystemC e totalmente compatíveis com o padrão IEEE™1666. É claro que além das extensões já elaboradas o usuário tem a liberdade de criar suas próprias bibliotecas, reunindo muitas vezes comportamentos similares dentro de uma API, por exemplo. Nos itens a seguir as extensões propostas pelos desenvolvedores são resumidamente apresentadas.

- TLM é uma metodologia de modelagem de sistemas que fornece um *framework* para desenvolvimento de plataformas virtuais, análise de performance, projeto a nível de arquitetura de sistemas e verificação funcional de hardware (MONTOREANO, 2007). Um dos conceitos principais da abordagem TLM é evitar descrições detalhadas em fases iniciais de desenvolvimento, e assim, tornar o modelo mais simples e compacto possível, acelerando simulação, por exemplo (RIGO; AZEVEDO; SANTOS, 2011). A metodologia TLM foi adicionada ao padrão IEEE™ e atualmente é bastante explorada na literatura e na indústria de forma geral;
- Para lidar com a crescente interação entre domínios analógicos e digitais, em sistemas *Embedded Analog/Mixed-Signal* (E-AMS) principalmente, foi proposto pela Accellera Systems Initiative a extensão AMS, construída sobre SystemC. Essa extensão fornece os meios para modelagem abstrata

de elementos mistos (analógicos e digitais) através de vários MoCs[1] e algumas semânticas adicionais. Como discutido anteriormente, apesar de SystemC ser uma excelente alternativa para modelagem de sistemas a nível de arquitetura, quando se trata de modelagem de algoritmos de controle, ou de elementos que apresentam comportamento analógico, SystemC por si só não é capaz de lidar com esse tipo de abstração. Justamente para preencher essa lacuna a extensão AMS foi desenvolvida, abrangendo assim, áreas que antes eram restritas a *Model-Based Design*. Atualmente uma implementação "*proof-of-concept*", desenvolvida pela empresa Fraunhofer IIS, é distribuída com uma licença Apache. Vários casos de sucesso de aplicação são apresentados em (OSCI, 2011)

- Uma das primeiras extensões elaboradas não foi para adicionar elementos para desenvolvimento, mas sim para realizar tarefas típicas de verificação. A extensão *SystemC Verification* (SCV) traz para SystemC a capacidade de elaborar *testbenches* mais sofisticados, contemplando múltiplos elementos para geração de estímulos e verificação dos resultados, como por exemplo randomização, restrições e conexão com linguagens de descrição de hardware como VHDL e Verilog. Esses mecanismos são implementados em formato de uma *Application Program Interface* (API), em que a principal função é evitar reimplementação dos mesmos mecanismos;

- CCI é uma extensão recentemente proposta para facilitar a troca de informação entre os modelos implementados e as ferramentas desenvolvidas por empresas EDA. Basicamente essa extensão propõe uma interface padrão, que irá permitir novas funcionalidades principalmente para depuração e análise dos modelos.

A.2 Exemplos de Modelo em SystemC

A criação de um modelo em SystemC pode ser realizada em diferentes níveis de abstração, considerando a descrição do comportamento e principalmente os detalhes de comunicação entre módulos/processos. Tratando-se de ISS geralmente os modelos são classificados quanto a representação do tempo

[1] Em SystemC-AMS foram introduzidos três novos modelos de computação: *Timed Data Flow* (TDR), usado para algoritmos de processamento como em comunicações (a modelagem é realizada de forma arquitetural ou funcional); *Linear Signal Flow* (LSF) permite a modelagem de sistemas de controle ou filtros com elementos para representação no domínio da frequência; *Electrical Linear Networks* (ELN) utilizado para modelagem de elementos elétricos, representados como sinais contínuos de tensão e corrente (BARNASCONI, 2010).

utilizada. Na Figura 26 são mostrados as principais classes de modelos. Evidentemente o modelo que toma um número maior de instruções para simular uma única instrução da arquitetura modelada, acaba sendo o menos eficiente em tempo de execução. Como pode ser visto na Figura 27 os modelos que possuem mais informações com relação a microarquitetura, são mais precisos para determinação do tempo de execução, porém, tem uma performance reduzida durante execução, exceto quando executados diretamente em *hardware* como em FPGA, por exemplo.

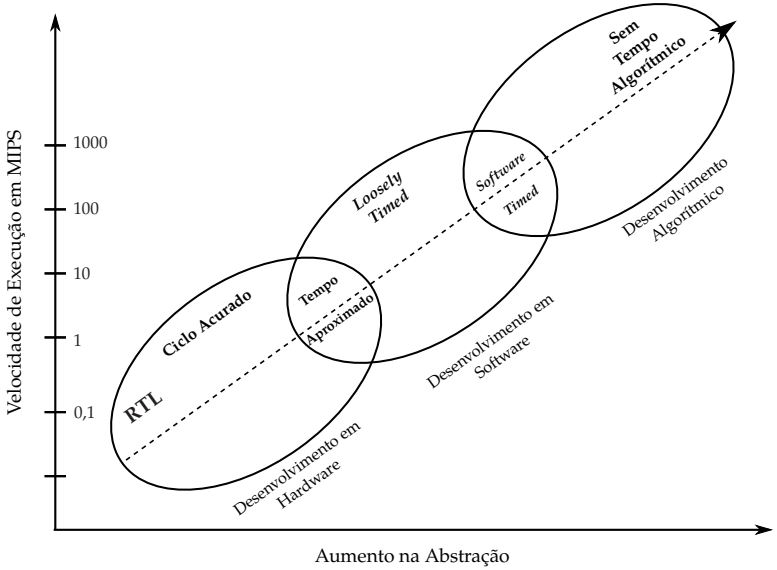

Figura 26 – Classificação Típica de ISSs Desenvolvidos em SystemC. Adaptado de (AARNO; ENGBLOM, 2014).

Um "esqueleto" geral de um ISS em SystemC, do tipo interpretador (a execução de uma instrução é modelada da mesma forma que seria feito em *hardware*, busca, decodifica, executa e armazena) é apresentado por Rigo, Azevedo e Santos (2011) e adaptado para a esse contexto no programa 11. Nesse caso a arquitetura modelada é do microcontrolador MSP430 da Texas Instruments™.

Na linha 5 do Programa 11 a classe que contém o simulador é instanciada. Linha 6 inicializa o processo em SystemC, que recebe parâmetros da linha de comando, como por exemplo o caminho para o executável a ser simulado. Linha 7 ativa o servidor de depuração que é implementado internamente ao simulador. Linha 10 inicia a simulação, a execução a partir desse ponto é

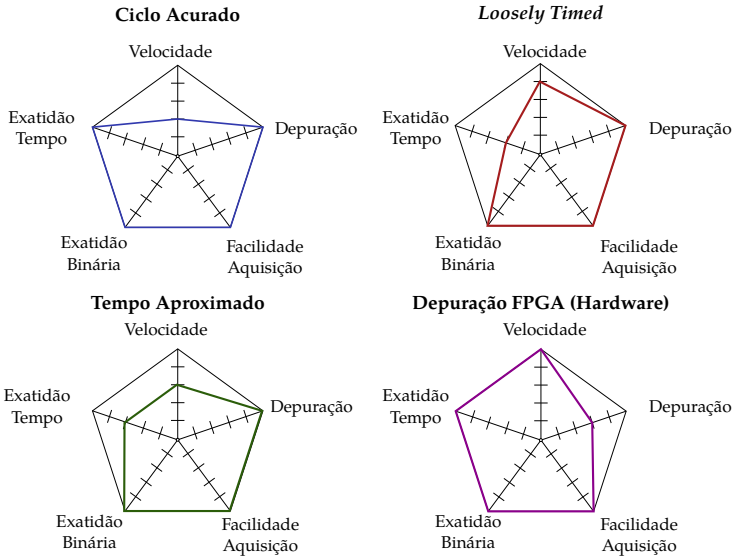

Figura 27 – Comparação entre as Abstrações de Modelagem de ISSs. Adaptado de (CHRISTIAN, 2010).

Programa 11 "Esqueleto" de um ISS em SystemC.

```
1       int sc_main(int ac, char *av[])
2       {
3         // 1) Elaboração
4         //!  ISA simulator
5         msp430 msp430_proc1("msp430");
6         msp430_proc1.init(ac, av);
7         msp430_proc1.enable_gdb(); // Interface de Depuração
8         .
9         .
10        // 2) Inicia Simulação
11        sc_start();
12        .
13        .
14        // 3) Depuração
15        msp430_proc1.PrintStat();
16
17        return msp430_proc1.ac_exit_status;
18      }
19
```

determinada pelo OSCI *Kernel*. Por fim, na linha 13, se não há mais instruções a serem executadas, ou por alguma exceção de execução, são impressas algumas informações de depuração e o programa termina a execução.

Como pode ser percebido boa parte do código seria repetido para um ISS diferente, esse é o princípio explorado por algumas linguagens LDA

como o ArchC. A descrição do comportamento do simulador é mostrado no programa 13.

Programa 12 Declaração da Classe msp430.

```
1       #include "systemc.h"
2       #include "ac_module.H"    // Herda de sc_module
        .
        .
3       .
4       #include "msp430_parms.H" // Constantes
5       #include "msp430_arch.H"  // Define registradores, ordenação,
6                                 // tamanho da palavra, etc
7       #include "msp430_isa.H"   // Encapsula informações binárias
8                                 // (campos e formatos)
        .
        .
9       .
10      class msp430: public ac_module, public msp430_arch, public AC_GDB_Interface
          <msp430_parms::ac_word>  {
11        public:
          .
          .
12        .
13        unsigned bhv_pc; // Contador de Programa
          .
          .
14        .
15        msp430_parms::msp430_isa ISA;
16
17        //!Behavior execution method.
18        void behavior();
19
20        SC_HAS_PROCESS( msp430 );
21        //!Constructor.
22        msp430( sc_module_name name_ ): ac_module(name_), msp430_arch(),
23        ISA(*this) {
24          SC_THREAD( behavior );
25          bhv_pc = 0;
            .
            .
26          .
27        }
          .
          .
28        .
29        virtual void PrintStat();
30
31        void load(char* program);
32
33        void stop(int status = 0);
34
35        void enable_gdb(int port = 5000);
36
37        virtual ~msp430() {};
38
39      };
40
```

No Programa 12 é apresentado a descrição da classe que encapsula o comportamento do simulador. Uma parte dó código original foi omitido por questões de clareza. Nas linhas 1 a 7 são incluídos os arquivos adicionais, a descrição está no próprio código. Na linha 10 é declarada a classe msp430, que por herança herda as classes `ac_module`, `msp430_arch` e `AC_GDB_Interface`. A classe `ac_module` adiciona funcionalidades a classe padrão para definição

de módulos no SystemC, `sc_module`; `msp430_arch` contém informações gerais sobre a arquitetura; `AC_GDB_Interface` realiza a interface com o servidor `gdb`, através de métodos, dependentes da arquitetura, definidos pelo usuário. Na linha 13 é declarado o contador de programa. Linha 15 contém a declaração da arquitetura do conjunto instruções. O processo principal de execução do simulador é definido como `behavior`, mostrado na linha 18 do Programa 13. Esse processo é executado pelo SystemC *Kernel*. Por fim, nas linhas 29 a 37 são declarados os métodos para imprimir informações com relação ao simulador (`PrintStat()`); o método para carregar o binário em memória (`load`); um método para finalização da simulação (`stop`); método para realizar inicialização e habilitar o servidor de depuração (`enable_gdb`); e o destruidor (`~msp430`).

Programa 13 Implementação do comportamento do Simulador.

```
1       #include "msp430.H"
2
3       void msp430::behavior() {
4
5         unsigned ins_id;
          .
          .
6         .
7         for (;;) {
8           // Decodifica Instrução
            .
            .
9           .
10          switch (ins_id) {
11            case 1: // Instruction mov_48imm
12            // Executa Instrução mov_48imm
13            break;
              .
              .
14            .
15          }
16          // Atualiza PC
            .
            .
17          .
18          wait(1, SC_NS); // Suspende Processo por 1 nano segundo
            .
            .
19          .
20        } // for (;;)
21      } // behavior()
22
```

Esses conjuntos de "recortes" de código de um ISS em SystemC foram extraídos diretamente de um modelo gerado pelo ArchC. O tipo de modelagem realizado é *Loosely Timed*, ou seja, as informações de tempo são abstratas, basicamente cada instrução levaria 1 ns para ser executada. Essa representação de tempo é interna ao OSCI *Kernel* de simulação e basicamente é utilizada para poder haver vários simuladores (instâncias da mesma classe) ao mesmo tempo. Cada vez que a função `wait()` é chamada, o *Kernel* de simulação faz uma mudança de contexto para executar qualquer processo que esta pronto para

ser executado em uma lista.

Essa revisão básica sobre SystemC, juntamente com o exemplo, é somente uma demonstração dos conceitos e formato geral dos modelos utilizados. Por mais que muitos conceitos de orientação a objetos sejam utilizados para criação de modelos em SystemC, boa parte do código é consideravelmente idiomático e só faz sentido dentro do contexto de SystemC. Uma revisão profunda sobre SystemC está fora do contexto desse trabalho.

APÊNDICE B – PROGRAMA PARA GERAÇÃO DOS VALORES RANDÔMICOS

B.1 Valores Randômicos

Programa 14 Programa utilizado para gerar os valores randômicos antes da simulação.

```
1
2  /* boost random_demo.cpp profane demo
3   *
4   * Copyright Jens Maurer 2000
5   * Distributed under the Boost Software License, Version 1.0. (See
6   * accompanying file LICENSE_1_0.txt or copy at
7   * http://www.boost.org/LICENSE_1_0.txt)
8   *
9   * Id : random_demo.cpp607552010 – 03 – 2200 : 45 : 06Zsteven_watanabe
10  *
11  * A short demo program how to use the random number library.
12  * Adaptado por Rogério Paludo
13  */
14
15 #include <iostream>
16 #include <fstream>
17 #include <ctime>            // std::time
18
19 #include <boost/random/linear_congruential.hpp>
20 #include <boost/random/uniform_int.hpp>
21 #include <boost/random/uniform_real.hpp>
22 #include <boost/random/variate_generator.hpp>
23
24 // Sun CC doesn't handle boost::iterator_adaptor yet
25 #if !defined(__SUNPRO_CC) || (__SUNPRO_CC > 0x530)
26 #include <boost/generator_iterator.hpp>
27 #endif
28
29 #ifdef BOOST_NO_STDC_NAMESPACE
30 namespace std {
31   using ::time;
32 }
33 #endif
34
35 #define MIN_throttle 3.0F    //degress
36 #define MAX_throttle 900.0F
37 #define PREC_throttle 0.1F
38
39 #define MIN_speed 0.0F      //radians/sec
40 #define MAX_speed 620.2F
41 #define PREC_speed 0.1F
42
43 #define MIN_ego 0.0F        //volts
44 #define MAX_ego 12.2F
45 #define PREC_ego 0.05F
46
47 #define MIN_press 0.05F    // bar
48 #define MAX_press 1.95F
49 #define PREC_press 0.001F
50
51 // This is a typedef for a random number generator.
52 // Try boost::mt19937 or boost::ecuyer1988 instead of boost::minstd_rand
53 typedef boost::minstd_rand base_generator_type;
54
55 using namespace std;
```

Programa 15 Programa utilizado para gerar os valores randômicos antes da simulação [Continuação].

```
1  int main()
2  {
3    // Define a random number generator and initialize it with a reproducible
4    // seed.
5    // (The seed is unsigned, otherwise the wrong overload may be selected
6    // when using mt19937 as the base_generator_type.)
7    base_generator_type generator(static_cast<unsigned int>(std::time(0)));
8
9    //  std::cout << "10 samples of a uniform distribution in [0..1):\n";
10
11   // Define a uniform random number distribution which produces "double"
12   // values between 0 and 1 (0 inclusive, 1 exclusive).
13   boost::uniform_real<> throttle_dist(MIN_throttle, MAX_throttle);
14   boost::uniform_real<> speed_dist(MIN_speed, MAX_speed);
15   boost::uniform_real<> ego_dist(MIN_ego, MAX_ego);
16   boost::uniform_real<> press_dist(MIN_press, MAX_press);
17
18   boost::variate_generator<base_generator_type&, boost::uniform_real<> >
        uni_throttle(generator, throttle_dist);
19   boost::variate_generator<base_generator_type&, boost::uniform_real<> >
        uni_speed(generator, speed_dist);
20   boost::variate_generator<base_generator_type&, boost::uniform_real<> >
        uni_ego(generator, ego_dist);
21   boost::variate_generator<base_generator_type&, boost::uniform_real<> >
        uni_press(generator, press_dist);
22
23   std::cout.setf(std::ios::fixed);
24   // You can now retrieve random numbers from that distribution by means
25   // of a STL Generator interface, i.e. calling the generator as a zero-
26   // argument function.
27   for(int i = 0; i < 100000; i++) {
28 #if 0
29     cout << "throttle: " << uni_throttle() << "\n"
30    << "speed: " << uni_speed() << "\n"
31    << "ego: " << uni_ego() << "\n"
32    << "press: " << uni_press() << "\n" << endl;
33 #endif
34     cout << uni_throttle() << ','
35    << uni_speed()    << ','
36    << uni_ego()      << ','
37    << uni_press()    << endl;
38   }
39   return 0;
40 }
```

APÊNDICE C – SIMBOLOGIA PARA OS FLUXOGRAMAS UTILIZADOS

C.1 Simbologia Fluxogramas

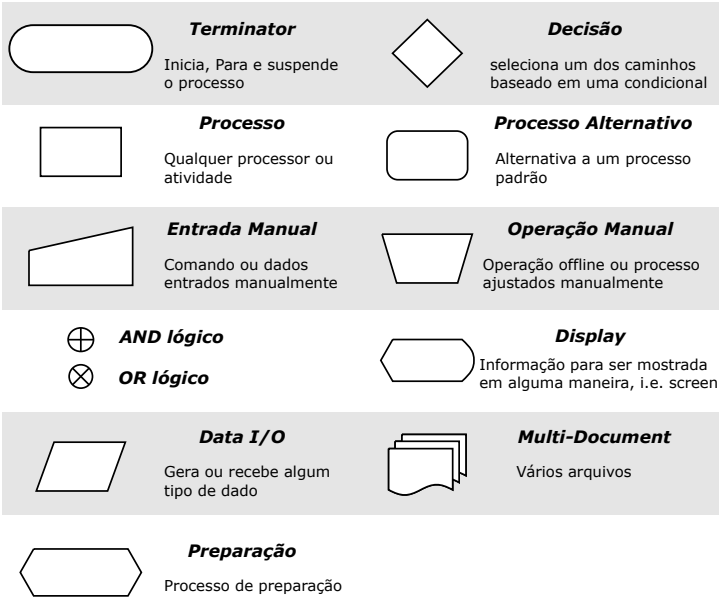

Figura 28 – Simbologia utilizada para os fluxogramas criados. Adaptado de IBM (1970).

APÊNDICE D – PUBLICAÇÕES

D.1 Publicações

PALUDO, R.; LETTNIN, D. A methodology for early functional verification of embedded software combining virtual platforms and bounded model checking. In: Test Symposium (LATS), 2016 17th Latin-American. [S.l.: s.n.], 2016. *A ser publicado.*

ZIJLSTRA, D.; PALUDO, R.; LETTNIN, D. Virtual Satellite Platform of an On-board Computer for Space Applications. In: International Academy of Astronautics: small satellites. Paris, França: International Academy of Astronautics, 2016. (IAA Proceedings Series). *Apresentado dia 29 de Março de 2016.*